Języki programowania

Notatki do wykładu „Programowanie” dla studentów I roku dziennych studiów magisterskich kierunku Informatyka na Uniwersytecie Wrocławskim

Języki programowania

ToMasz Wierzbicki

Instytut Informatyki Uniwersytetu Wrocławskiego

ToMasz Wierzbicki
Uniwersytet Wrocławski
Instytut Informatyki
Zakład Języków Programowania
Przesmyckiego 20
51–151 Wrocław
http://www.ii.uni.wroc.pl/~tomasz/

Języki programowania
Notatki do wykładu „Programowanie” dla studentów I roku dziennych studiów magisterskich kierunku Informatyka na Uniwersytecie Wrocławskim
Słowa kluczowe: **języki programowania**, **teoria programowania**
Symbol Uniwersalnej Klasyfikacji Dziesiętnej: **004.43**
2000 Mathematics Subject Classification: **68N15**

Przygotowanie notatek i druk 30 egzemplarzy na potrzeby Biblioteki Instytutu Informatyki Uniwersytetu Wrocławskiego sfinansowano ze środków projektu **Tempus JEP 12266/97** w ramach programu Phare sponsorowanego przez Unię Europejską

Adres elektronicznej wersji notatek: http://www.ii.uni.wroc.pl/~tomasz/jp/

Skład w LaTeX-u i rysunki w XFig-u: autor
Korekta: Oskar Miś
Czcionka: Times Roman 10pt
Data utworzenia dokumentu: 17 stycznia 2001
Wydanie: 0.75

Cudakowi

There are so many programming languages in existence that it is a hopeless task to attempt to learn them all.

Christopher Strachey,
The Varietes of Programming Language,
1973

Spis treści

Przedmowa

Bieżąca wersja notatek obejmuje około 3/4 materiału wykładanego w ramach zajęć z „Programowania”, dlatego nazwałem ją wydaniem 0.75. Serdecznie dziękuję Kierownikom projektu *Tempus JEP 12266/97* oraz Jurkowi Marcinkowskiemu za zachętę do napisania niniejszych notatek (także finansową — przygotowanie i druk 30 kopii opłacono ze środków tego projektu). Jestem też wdzięczny Pawłowi Rychlikowskiemu, Pawłowi Rajbie i Pawłowi Rzechonkowi, którzy je przeczytali i wskazali wiele błędów.

W niektórych miejscach zamiast treści notatek występuje zwięzła lista zagadnień zatytułowana „do uzupełnienia”. Fragmenty te zostaną wypełnione tekstem w kolejnych wydaniach notatek. Aby ich brak był mniej dokuczliwy dla czytelnika bieżącego wydania, w takich miejscach podana jest zwykle „literatura zastępcza”, tj. lista kilku książek, które zawierają podobny materiał.

Autor

Rozdział 1

Wstęp

Do uzupełnienia: Znaczenie teorii języków programowania w informatyce.

Rozdział 2

Podstawy programowania w języku Standard ML ’97

Język ML został opracowany w latach 80-tych XX wieku w Edynburgu przez Robina Milnera i jego współpracowników. Początkowo był projektowany jako metajęzyk systemu dowodzenia twierdzeń *LCF* (ang. *MetaLanguage*, stąd nazwa ML). Wkrótce jednak stał się językiem programowania ogólnego przeznaczenia. Robin Milner, Mads Tofte i Robert Harper opublikowali w 1990 pierwszy standard języka [116, 117], zwanego od tej pory Standard ML-em, lub krócej SML-em. Twórcy innych dialektów, którzy nie podporządkowali się standardowi, zaczęli na bazie ML-a rozwijać własne języki (spośród nich najbardziej znany jest Objective Caml). Aktualna definicja języka [118], opracowana wspólnie przez wspomnianych autorów i Davida MacQueena, pochodzi z roku 1997. Jeżeli zachodzi potrzeba rozróżnienia obu standardów, mówi się o SML-u ’90 i SML-u ’97.

Standard ML zawiera wiele konstrukcji spotykanych w nowoczesnych językach programowania, których brakuje w popularnych językach, takich jak Pascal i C: ścisły statyczny system typów zapewniający pełne bezpieczeństwo w sensie ochrony pamięci i innych błędów czasu wykonania, automatyczne zarządzanie pamięcią, kontrolę i rekonstrukcję typów, polimorfizm parametryczny, zarówno trwałe jak i ulotne struktury danych, w szczególności algebraiczne typy danych `datatype` i mechanizm wzorców do ich przetwarzania, wyjątki, typy abstrakcyjne, jeden z najbardziej rozbudowanych systemów modularyzacji i specyfikacji programów (zawierający moduły z parametrami), itd. Ma jednocześnie wygodne mechanizmy wspierające programowanie funkcjonalne i programowanie współbieżne. Jako jedyny używany w praktyce język programowania ma całkowicie formalną definicję matematyczną. Posiada również kilka efektywnych i stabilnych implementacji.

Bieżący rozdział, mający charakter zwięzłego podręcznika programisty, jest poświęcony przedstawieniu podstawowych konstrukcji SML-a. Więcej informacji można znaleźć w popularnych podręcznikach [135, 179, 50, 68, 31, 112, 23, 157, 154, 144].[1]

[1] Aktualną hipertekstową bibliografię SML-a (wraz z tekstami dostępnymi w wersji elektronicznej) można zna-

2.1. Praca z kompilatorem języka Standard ML

Najpopularniejszym kompilatorem języka Standard ML jest *Standard ML of New Jersey* (w skrócie SML/NJ) opracowany oryginalnie w Bell Laboratories i Princeton University, a obecnie rozwijany w ramach wspólnego projektu Bell Laboratories (Lucent Technologies), Princeton University, Yale University i AT&T Research. Kompilator jest dostępny za darmo pod adresem `http://cm.bell-labs.com/cm/cs/what/smlnj/`.

2.1.1. Uwagi dla użytkowników kompilatora SML/NJ

Kompilator SML/NJ, uruchamiany przeważnie poleceniem `sml`, pracuje w trybie interakcyjnym. System wypisuje znak zachęty (`-`) i czeka na wprowadzenie poleceń z klawiatury. Programista może wprowadzać tekst w wielu wierszach, na początku każdego następnego wiersza system drukuje znak kontynuacji (`=`).[2] Koniec tekstu wprowadzonego z klawiatury programista zaznacza średnikiem:

```
- 1 +
= 1;
val it = 2 : int
```

Podczas wprowadzania kolejnych wierszy kompilator analizuje tekst składniowo i sygnalizuje ewentualne błędy. Po zakończeniu wprowadzania tekstu poprawnego składniowo kompilator dokonuje kontroli typów, tłumaczy i optymalizuje program, a następnie go wykonuje. Kontrola typów może zakończyć się komunikatem o błędzie. Również w czasie wykonania programu obliczenie może zostać zerwane. Jeśli jednak wszystko przebiegnie zgodnie z planem, kompilator wypisuje wynik obliczenia i kolejny znak zachęty.

Krótkie, jednolinijkowe programy (w szczególności pojedyncze wyrażenia, które są najprostszymi programami) można wprowadzać wprost z klawiatury w opisany wyżej sposób. Dłuższe lepiej przygotować w pliku (np. `myprog.sml`). Do wczytania programu z pliku służy funkcja `use`, której jako parametr podajemy nazwę pliku ujętą w cudzysłowy:

```
- use "myprog.sml";
[opening myprog.sml]
      ...
val it = () : unit
```

W programie można umieszczać *komentarze*, ignorowane przez kompilator. Podobnie jak w Pascalu ogranicznikami komentarzy są `(*` i `*)` (nie można jednak używać Pascalowych komentarzy `{` i `}`). Komentarze można zagnieżdżać.

Przerwanie wykonania obliczeń (np. gdy nasz program się zapętlił) i powrót do głównej pętli interpretera osiąga się naciśnięciem klawisza `ctrl-C`. Klawisz `ctrl-Z` służy, jak wszędzie pod Unix-em, do chwilowego zatrzymania pracy systemu i powrotu do powłoki systemowej. Unix-owe polecenie `fg` pozwala powrócić do zatrzymanego systemu. Ostatniego

leźć pod adresem `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/literature.html`.

[2] Można oczywiście skonfigurować system tak, by drukował dowolne inne napisy.

polecenia można używać przy pracy na terminalu tekstowym, gdy na przemian chcemy edytować plik i go kompilować. Dużo wygodniej można pracować z edytorem `emacs`, który po załadowaniu odpowiednich makrodefinicji pozwala na kompilowanie edytowanego programu za pomocą naciśnięcia pojedynczego klawisza. Najwygodniej jednak pracować w kilku oknach i otworzyć osobne dla edytora i osobne dla systemu SML-a. Sesję zamyka się naciskając klawisz `ctrl-D`.

Odpowiedzi systemu podczas sesji interakcyjnej są dosyć lakoniczne. Aby nie zaśmiecać ekranu system ukrywa niektóre informacje, wypisując w ich miejscu znak `#`. Wpisanie dwu tajemniczych wierszy:

```
Compiler.Control.Print.printDepth := 10000;
Compiler.Control.Print.printLength := 10000;
```

...nie, nie spowoduje sformatowania twardego dysku, tylko sprawi, że system będzie wypisywał znacznie więcej informacji.

2.2. Predefiniowane typy danych

2.2.1. Liczby całkowite i rzeczywiste

W języku Standard ML '97 do reprezentowania liczb całkowitych przewidziano dwa typy danych: `int` (liczby całkowite ze znakiem) i `word` (liczby całkowite bez znaku). *Literały*[3] *całkowitoliczbowe* to niepuste ciągi cyfr dziesiętnych, domyślnie oznaczające liczby typu `int` w zapisie dziesiętnym. Można je poprzedzić przedrostkiem `0w` (na oznaczenie wartości typu `word`), `0x` (na oznaczenie zapisu szesnastkowego) lub `0wx` (na oznaczenie wartości typu `word` w zapisie szesnastkowym). Literały typu `int` (zarówno w zapisie dziesiętnym jak i szesnastkowym) można poprzedzić znakiem `~` oznaczającym liczbę ujemną (*uwaga:* w wielu językach używa się do tego celu znaku `-`). W zapisie szesnastkowym można używać zarówno małych, jak i wielkich liter `a` ÷ `f` na oznaczenie cyfr 10 ÷ 15. Dla przykładu:

literał	oznacza liczbę
`13`	13 typu `int`
`0x13`	19 typu `int`
`~0x13`	−19 typu `int`
`0w13`	13 typu `word`
`0wx13`	19 typu `word`
`~0w13`	jest błędny
`~0wx13`	jest błędny
`0xFF`	255 typu `int`
`0wx3fffffff`	$2^{32} - 1$ typu `word`

[3] Ciągi znaków programu reprezentujące stałe. O literałach mówimy więcej w rozdziale 3.

Literały zmiennopozycyjne typu `real` składają się z następujących części: opcjonalnie znaku `~` (oznaczającego, podobnie jak w literałach całkowitoliczbowych, wartość ujemną), niepustego ciągu cyfr oznaczającego część całkowitą, kropki po której następuje niepusty ciąg cyfr (część ułamkowa) i znaku `e` lub `E`, po którym następuje niepusty ciąg cyfr (wykładnik) poprzedzony opcjonalnie znakiem `~`. Część ułamkowa lub wykładnik mogą być pominięte, jednak nie obie jednocześnie. Dla przykładu literałami zmiennopozycyjnymi są `1.2`, `~1.2`, `~1e~1` i `~0.1`. Dwa ostatnie oznaczają tę samą liczbę. Napis `13` nie jest literałem zmiennopozycyjnym, ponieważ nie zawiera ani części ułamkowej, ani wykładnika.

W odróżnieniu od wielu języków programowania, w języku Standard ML nie można w jednym wyrażeniu używać wartości różnych typów numerycznych, np. wyrażenia `1+2` i `2.5+4.7` są poprawne (i pierwsze jest typu `int`, drugie zaś typu `real`), zaś wyrażenie `1+2.3` nie jest poprawne, ponieważ `1` jest literałem całkowitoliczbowym a nie zmiennopozycyjnym (to zagadnienie jest omówione dokładniej w podrozdziale 2). Ostatnie wyrażenie należy zapisać w postaci `1.0+2.3`. Do jawnej zamiany reprezentacji liczb typu `int` na typ `real` służy funkcja `real`, do zamiany w przeciwną stronę — funkcje `trunc` (odrzucająca część ułamkową), `floor` (zwracająca największą liczbę całkowitą nie większą od podanej liczby zmiennopozycyjnej), `ceil` (zwracająca najmniejszą liczbę całkowitą nie mniejszą od podanej liczby zmiennopozycyjnej) i `round` (zwracająca „najbliższą” liczbę całkowitą). Dostępne operatory numeryczne, ich priorytety i łączliwość[4] opisano w tablicy 2.1 na sąsiedniej stronie. Operacje `div` i `mod` mają następujące znaczenie: a `div` $b = q$ i a `mod` $b = r$, jeśli $a = b * q + r$ oraz

$$\begin{aligned} 0 \leq r < b \quad &\text{gdy} \quad b > 0 \\ b < r \leq 0 \quad &\text{gdy} \quad b < 0 \end{aligned}$$

Zatem np. `7 mod ~3 = ~2`. Zauważmy, że (zgodnie ze standardem IEEE) dla liczb zmiennopozycyjnych nie zdefiniowano relacji równości.[5]

Co prawda oba argumenty operatora takiego jak `+` muszą być tego samego typu, jednak może on działać na danych różnych typów (`int`, `word`, `real`). O takich operatorach mówi się, że są *przeciążone* (o przeciążaniu operatorów jest mowa w podrozdziale 9.3.2). Z każdym operatorem jest związany domyślny typ jego argumentów, wybierany wówczas, gdy z kontekstu użycia operatora nie wynika, jakiego są one typu. Dla operatora `+` domyślnym typem jego argumentów jest `int`.

[4]Im wyższy priorytet, tym operator wiąże *silniej*, dzięki temu można, podobnie jak w matematyce, opuszczać nawiasy: ponieważ operator `*` wiąże silniej (ma wyższy priorytet) niż `+`, to wyrażenie `4+2*3` oznacza `4+(2*3)` a nie `(4+2)*3`. Łączliwość operatora pozwala opuszczać nawiasy w wyrażeniach takich jak `2+3+4`. Ponieważ operator `+` łączy w lewo, ostatnie wyrażenie oznacza `(2+3)+4`. Gdyby łączył w prawo, oznaczałoby `2+(3+4)`. Operatory *prefiksowe* pisze się przed ich argumentem. Wiążą one silniej niż jakiekolwiek operatory *infiksowe* (pisane pomiędzy argumentami), np. `~x+y` oznacza `(~x)+y`, a nie `(~x+y)`. O operatorach, ich priorytetach i łączliwości mówimy więcej w podrozdziałach 3.2.2 i 3.9.1.

[5]Błędy zaokrągleń powodują, że porównywanie jest często bezsensowne, np. w starej implementacji *SML/NJ, version 0.93*, w której było dostępne porównywanie liczb rzeczywistych, system z uporem zaprzeczał, jakoby `1.2-1.0=0.2`.

operacja	operator	priorytet	łączliwość	dostępne dla typu		
				`int`	`word`	`real`
zmiana znaku	`~`	prefiksowy		•		•
wartość bezwzględna	`abs`	prefiksowy		•		•
mnożenie	`*`	7	w lewo	•	•	•
dzielenie całkowite	`div`	7	w lewo	•	•	
reszta z dzielenia	`mod`	7	w lewo	•	•	
dzielenie	`/`	7	w lewo			•
dodawanie	`+`	6	w lewo	•	•	•
odejmowanie	`-`	6	w lewo	•	•	•
relacje porządku	`<`, `<=`, `>`, `>=`	4	w lewo	•	•	•
relacje równości	`=`, `<>`	4	w lewo	•	•	

Tablica 2.1. Operacje na danych numerycznych w SML-u

2.2.2. Wartości logiczne

Typ `bool` ma dwie wartości reprezentowane przez *literały logiczne* `true` i `false`. *Wyrażenie warunkowe* (podobnie jak `?:` w C)

$$\texttt{if } b \texttt{ then } x \texttt{ else } y$$

ma następujące znaczenie: najpierw jest obliczane wyrażenie b, a następnie, w zależności od jego wartości — jedno z wyrażeń x bądź y. Wyrażenia

$$b_1 \texttt{ andalso } b_2 \quad \equiv \quad \texttt{if } b_1 \texttt{ then } b_2 \texttt{ else false}$$
$$b_1 \texttt{ orelse } b_2 \quad \equiv \quad \texttt{if } b_1 \texttt{ then true else } b_2$$

to koniunkcja i alternatywa wyrażeń boolowskich, przy czym zawsze jako pierwsze obliczane jest wyrażenie b_1, a b_2 jedynie wówczas, gdy wartość b_1 nie przesądza wyniku. Funkcja `not` oblicza negację swojego argumentu.

2.2.3. Krotki i rekordy

Krotki (ang. *tuple*) zapisuje się oddzielając ich elementy przecinkami i ujmując całość w nawiasy okrągłe, np.

```
(1,true)
```

jest parą złożoną z liczby całkowitej `1` i wartości boolowskiej `true`. Krotki są szczególnym przypadkiem *rekordów*. W rekordzie pola posiadają etykiety, np.

```
{dzien=22,miesiac=1,rok=1969}
```

jest rekordem o trzech polach typu `int` o etykietach `dzien`, `miesiac` i `rok`. Etykietami pól mogą być identyfikatory lub liczby naturalne. Krotka jest rekordem o polach nazywanych kolejnymi liczbami naturalnymi. Do selekcji pól rekordu służy specjalna konstrukcja postaci `#` $l\ r$, gdzie l jest etykietą, a r — rekordem, np. wyrażenie

```
#miesiac {dzien=22,miesiac=1,rok=1969}
```

ma wartość `1`. Ponieważ wygodniej jest korzystać z *dopasowania wzorca* (o którym mówimy w podrozdziałach 2.7 i 2.9), konstrukcja `#` $l\ r$ jest używana dosyć rzadko.

2.2.4. Typ unit

Typ `unit` ma podobne zastosowanie jak typ `void` w C — wartość tego typu zwracają w wyniku funkcje, które faktycznie nie dostarczają żadnej wartości (powodują tylko skutki uboczne). Ten typ posiada dokładnie jeden element, który można zapisać w SML-u na dwa sposoby: `()` lub `{}`. Te dziwne oznaczenia biorą się stąd, że na element typu `unit` można patrzeć jak na zeroelementowy rekord `{}` lub zeroelementową krotkę `()`. Istotnie, produkt zera egzemplarzy jakiegokolwiek zbioru jest zbiorem dokładnie jednoelementowym!

2.2.5. Listy

Listy to kolekcje elementów tego samego typu, ale zmiennej długości. Listy zapisuje się oddzielając ich elementy przecinkami i ujmując całość w nawiasy kwadratowe, np.

```
[1,2,3]
```

Konstruktorem listy (operatorem „budującym" listę, por. podrozdział 2.7) jest łączący w prawo operator `::` o priorytecie 5 dołączania głowy do listy i konstruktor listy pustej `nil`. Napis z użyciem nawiasów jest jedynie wygodnym skrótem notacyjnym: `[]` jest równoważne `nil`, zaś lista `[1,2,3]`, to tak na prawdę `1::(2::(3::nil))`. Na listach jest zdefiniowany operator spinania list `@` (łączący w prawo, o priorytecie 5) oraz funkcje: `rev` odwracająca listę, `null` sprawdzającą, czy lista jest pusta, `length` ujawniającą długość (liczbę elementów) listy, oraz `hd` i `tl` ujawniające odpowiednio głowę (pierwszy element) i ogon (listę wszystkich poza pierwszym elementem) listy. Ponieważ w SML-u powszechnie używa się dopasowania wzorca (o którym mówimy w podrozdziałach 2.7 i 2.9), dwie ostatnie funkcje są wykorzystywane bardzo rzadko, a ich użycie bez uzasadnionej potrzeby jest uważane za wyjątkowo nieeleganckie. Standardowo dostępne są także funkcje:

$$\begin{aligned}
\texttt{map}\ f\ [x_1,\ldots,x_n] &= [f\,x_1,\ldots,f\,x_n] \\
\texttt{app}\ f\ [x_1,\ldots,x_n] &= (f\,x_1;\ldots;f\,x_n;()) \\
\texttt{foldl}\ f\ c\ [x_1,\ldots,x_n] &= f(x_n,\ldots,f(x_2,f(x_1,c))\ldots) \\
\texttt{foldr}\ f\ c\ [x_1,\ldots,x_n] &= f(x_1,f(x_2,\ldots f(x_n,c)\ldots))
\end{aligned}$$

(o operatorze złożenia sekwencyjnego `;` występującym w opisie funkcji `app` jest mowa w podrozdziale 2). Funkcja `map` tworzy listę wyników aplikacji funkcji f do elementów $x_1,\ldots,x_n$, np. wartością wyrażenia

`\a`	alarm (BEL)
`\b`	cofanie (BS)
`\f`	nowa strona (FF)
`\n`	nowy wiersz (LF)
`\r`	powrót karetki (CR)
`\t`	tabulacja pozioma (HT)
`\u`*hhhh*	znak o kodzie *hhhh*, gdzie *hhhh* jest ciągiem czterech cyfr szesnastkowych
`\v`	tabulacja pionowa (VT)
`\\`	znak odwróconego ukośnika (\)
`\"`	znak cudzysłowu (")
`\`*ddd*	znak o kodzie *ddd*, gdzie *ddd* jest ciągiem trzech cyfr dziesiętnych reprezentującym liczbę z przedziału $0 \div 255$
`\`*białe–znaki*`\`	*białe–znaki* (spacje, znaki tabulacji, nowego wiersza, nowej strony itp.) wraz z otaczającymi je odwróconymi ukośnikami są ignorowane; pozwala to wprowadzić napis dłuższy niż jeden wiersz
`\^`*c*	znak sterujący o kodzie (kod znaku *c*) $-$ 64

Tablica 2.2. Specjalne sekwencje znaków

```
map abs [1, ~2, ~3, 2]
```

jest lista `[1,2,3,2]`. Efekt obliczenia `app` *f l* polega na wywołaniu skutków ubocznych aplikacji funkcji *f* do kolejnych elementów listy *l*, np.

```
app print ["Cudak"," ","jest"," ","wielki","!","\n"]
```

powoduje wypisanie na ekranie tekstu

```
Cudak jest wielki!
```

Funkcje `foldl` i `foldr` „zwijają" listę, odpowiednio od lewej i od prawej (przykłady ich użycia znajdują się w podrozdziale 2.8).

2.2.6. Dane znakowe

Do reprezentowania znaków służą elementy typu `char`. *Literały znakowe* mają postać `#"`*c*`"`, gdzie *c* jest albo pojedynczym znakiem, albo sekwencją specjalną opisaną w tablicy 2.2. Funkcja `chr` tworzy znak o podanym kodzie (w postaci liczby typu `int` z przedziału $0 \div 255$), funkcja `ord` ujawnia kod podanego znaku. Operacje porównania znaków `<`, `>`, `<=`, `>=`, `=` i `<>` (łączące w lewo, o priorytecie 4) porównują ich kody.

2.2.7. Łańcuchy (dane napisowe)

Literał łańcuchowy to dowolny ciąg znaków (różnych od \, " i znaku nowego wiersza) oraz sekwencji specjalnych opisanych w tablicy 2.2 na poprzedniej stronie. Do przetwarzania łańcuchów służy operator spinania ^ i funkcje `explode` i `implode` zamieniające łańcuch na listę znaków i odwrotnie, `size` ujawniająca długość łańcucha, `str` zamieniająca znak na jednoznakowy łańcuch, `concat` spinająca listę łańcuchów w jeden łańcuch i `substring` (s, i, n) wyznaczająca fragment łańcucha s od pozycji i (licząc od zera) i długości n znaków. Na łańcuchach zdefiniowano operacje leksykograficznego porównania kodów ich znaków <, >, <=, >=, = i <> (łączące w lewo, o priorytecie 4). Funkcja `print` wypisuje łańcuch do standardowego strumienia wyjściowego.

2.2.8. Biblioteka funkcji standardowych

Wszystkie opisane wyżej wartości i działające na nich funkcje są dostępne w tzw. środowisku pierwotnym, tj. są rozpoznawane w programie przez kompilator natychmiast po uruchomieniu systemu, bez dodatkowych zabiegów. Ponadto w systemie SML-a jest dostępna spora biblioteka funkcji standardowych, umieszczonych w tzw. *strukturach* (odpowiednikach *modułów* lub *pakietów* w innych językach, por. rozdział 15), tzw. *Standard ML Basis Library*. Do tych funkcji należy się odwoływać poprzedzając ich nazwę nazwą odpowiedniej struktury, np. `Math.sin` jest nazwą funkcji `sin` umieszczonej w strukturze `Math`, zatem `Math.sin 5.0` jest poprawnym wyrażeniem, podczas gdy wyrażenie `sin 5.0` spowoduje błąd kompilacji z komunikatem „nieznany identyfikator `sin`”. Kompletny hipertekstowy opis biblioteki standardowej jest dostępny pod adresem `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/index.html`.

2.3. Funkcje jako wartości

Opisane w podrozdziale 2.2 wartości różnych typów i operatory standardowe pozwalają budować najprostsze *wyrażenia*. Bardziej skomplikowane wyrażenia mogą zawierać funkcje, nie tylko standardowe, lecz również definiowane przez użytkownika. W SML-u funkcje są traktowane tak samo, jak inne wartości: mogą być elementami krotek i list, można je przekazywać jako parametry innym funkcjom i mogą być przez takie funkcje zwracane w wyniku. Do operowania na funkcjach służą w SML-u dwie podstawowe konstrukcje: *abstrakcja funkcyjna* wykorzystywana do budowania nowych funkcji i *aplikacja funkcji do argumentu*, pozwalająca użyć danej funkcji.

2.3.1. Abstrakcja funkcyjna

Często w matematyce pisze się „funkcja $x^2 + 1$”. Zwróćmy uwagę, że jest to niepoprawne: podnosić do kwadratu umiemy liczby, zatem x jest liczbą (dokładniej: zmienną, która może przyjmować wartości liczbowe), kwadrat jakiejś liczby jest liczbą, gdy dodamy do niej jedynkę dalej będzie liczbą. Zatem $x^2 + 1$ jest liczbą (dokładniej: wyrażeniem przyjmują-

cym wartości liczbowe) a nie funkcją.[6] Być może kilka linijek wcześniej napisano „niech x będzie równe 5". Wówczas $x^2 + 1$ jest po prostu równe 26. Puryści piszą więc „funkcja $x \longmapsto x^2 + 1$". Wówczas wystąpienie x-a pod kwadratem jest *związane* jego wystąpieniem przed strzałką, a cały napis oznacza „odwzorowanie, które dowolnemu x-owi przyporządkowuje wartość $x^2 + 1$". Podobny sposób pisania przyjęto w SML-u: jeśli E jest wyrażeniem, a x — zmienną (być może występującą w tym wyrażeniu), to następujące wyrażenie, zwane *abstrakcją funkcyjną*:

`fn` x `=>` E

oznacza funkcję, która argumentowi x przyporządkowuje wartość E. Np. wyrażenie opisujące funkcję z ostatniego przykładu zapiszemy w SML-u jako

```
fn x => x*x + 1
```

W odróżnieniu od Pascala i C możemy zatem w SML-u definiować *funkcje anonimowe*, tj. wyrażenia, których wartościami są funkcje, bez konieczności nadawania nazwy definiowanym funkcjom. Zwróćmy uwagę, że nic nie stoi na przeszkodzie by napisać

```
fn x => fn y => x*y + 1
```

Ostatnie wyrażenie oznacza funkcję, która argumentowi `x` przyporządkowuje wartość będącą funkcją, która argumentowi y przyporządkowuje wartość `x*y + 1`.

Pojęcie *zmiennej wolnej* i *związanej* w SML-owym wyrażeniu ma taki sam sens jak w matematyce. Tu „kwantyfikatorem" wiążącym zmienne (podobnie jak $\forall$ i $\exists$ w logice, czy zapis $\int \cdots dx$ w analizie) jest słowo kluczowe `fn`. Zmienna (identyfikator) x jest *związana* w wyrażeniu `fn` x `=>` E, jeśli natomiast występuje poza zasięgiem konstrukcji `fn` x `=>`, wówczas jest *wolna*. W wyrażeniu `fn x => x+y` zmienna `x` jest związana, zaś `y` — wolna. W wyrażeniu `x (fn x => x)` pierwsze wystąpienie zmiennej `x` jest *wolne*, drugie *wiążące*, a trzecie *związane*. Program przedstawiony do kompilacji nie może zawierać zmiennych wolnych (w przeciwnym przypadku pojawia się komunikat „niezadeklarowany identyfikator").

2.3.2. Aplikacja funkcji do argumentu

Jeśli napiszemy dwa wyrażenia (przy czym wartością pierwszego musi być funkcja) jedno po drugim, to otrzymamy wyrażenie zwane *aplikacją funkcji do argumentu*, np.

```
rev [1,2]
```

jest aplikacją funkcji `rev` do argumentu `[1,2]` i ma wartość `[2,1]`, a

```
(fn x => x*x + 1) (1+3)
```

[6] Dokonaliśmy tutaj *rekonstrukcji typów* o której mówimy więcej w podrozdziale 9.2.

jest aplikacją funkcji `fn x => x*x + 1`, do argumentu `1+3` i ma wartość `17`. Aplikacja wiąże w lewo i silniej niż abstrakcja oraz wszelkie operatory infiksowe, zatem

$$\texttt{fn } x \texttt{ => } E_1\ E_2\ \ldots\ E_n$$

oznacza

$$\texttt{fn } x \texttt{ => } (\ldots(E_1\ E_2)\ \ldots\ E_n)$$

a np.

```
f rev [1,2] @ rev [3,4]
```

oznacza

```
((f rev) [1,2]) @ (rev [3,4])
```

Dowolne wyrażenie ujęte w nawiasy dalej jest wyrażeniem o tej samej wartości. Nawiasów należy jednak używać z umiarem. W innych językach programowania przyzwyczailiśmy się otaczać nawiasami argument funkcji. Nic nie stoi na przeszkodzie, by i w SML-u napisać

```
rev ([1,2])
```

W SML-u możemy ponadto napisać

```
rev [1,2] lub (rev) [1,2]
```

i wszystkie trzy wyrażenia mają takie samo znaczenie.

2.4. Nadanie nazwy

Pisanie programu w postaci jednego, wielkiego wyrażenia byłoby wyjątkowo niepraktyczne. SML pozwala nadać dowolnemu wyrażeniu nazwę i posługiwać się nią w dalszych obliczeniach. Składnia deklaracji *nadania nazwy* (ang. *value binding*) jest następująca:

$$\texttt{val } x \texttt{ = } E$$

gdzie x jest identyfikatorem, a E — nazywanym wyrażeniem. Mówi się, że nazwa (identyfikator) x zostaje *związana (bound)* z wartością wyrażenia E (które najpierw zostaje obliczone). Dla przykładu możemy odbyć z komputerem następującą rozmowę:

```
- val f = fn x => x*x + 1;
val f = fn : int -> int
```

Kompilator odpowiada, że funkcję `fn x => x*x + 1` (która, jak sprytnie wykombinował, bierze argument typu `int` i dostarcza wartości typu `int`) nazwano `f`. Od tej chwili, aż do zakończenia sesji[7] możemy używać identyfikatora `f` w wyrażeniach i ich znaczenie jest dokładnie takie samo, jak gdybyśmy wstawili w miejsce `f` wyrażenie `fn x => x*x + 1`, np.

[7] Istnieją metody zachowania stanu sesji i odtworzenia go później.

```
- f 12;
val it = 145 : int
```

W dowolnej chwili możemy powtórnie użyć tego samego identyfikatora do nazwania innej wartości. Poprzednia wartość staje się wówczas niedostępna. Jeśli w innych miejscach programu odwoływano się do identyfikatora przed przedefiniowaniem go, to te odwołania dotyczą „starej" wartości z nim związanej:

```
- val x = 7;
val x = 7 : int
- fun f y = x+y+5;
val f = fn : int -> int
- f 3;
val it = 15 : int
- val x = 12;
val x = 12 : int
- f 3;
val it = 15 : int
- val f = fn y => x+y+5;
val f = fn : int -> int
- f 3;
val it = 20 : int
```

W chwili zadeklarowania funkcji `f` identyfikator `x` jest związany z liczbą 7, zatem

```
(fn y => x+y+5) 3  =  15
```

Jeśli wykorzystamy `x` ponownie, tym razem do nazwania liczby 12, to wartość związana z `f` nie ulegnie zmianie. Ten rodzaj wiązania nazwy z wartością, to tzw. *wiązanie statyczne*. Zatem nadanie nazwy w SML-u *nie jest przypisaniem wartości zmiennej*, zachowuje się raczej jak deklaracja `const` w Pascalu.

Program w SML-u można traktować jako ciąg deklaracji. W głównej pętli interpretera po symbolu zachęty możemy wpisać wyrażenie. System odpowiada jednak tak, jakby to było nadanie wartości tego wyrażenia nazwy `it`:

```
- 5;
val it = 5 : int
```

Istotnie, z nazwy `it` możemy korzystać w dalszych obliczeniach:

```
- 2*it;
val it = 10 : int
- 2*it;
val it = 20 : int
```

Nazwy `it` lepiej nie przedefiniowywać samemu (może to skonfudować zarówno programistę jak i kompilator).

2.5. Deklaracje funkcji i rekursja

Identyfikator występujący po słowie `val` nie jest dostępny w wyrażeniu, którego wartość ma nazwać, dlatego np. próba zdefiniowania funkcji silnia w poniższy sposób nie powiedzie się:

```
- val silnia = fn n =>
=        if n <= 0
=           then 1
=           else n * silnia (n-1);
Error: unbound variable or constructor: silnia
```

Aby dać znać kompilatorowi, że definiujemy funkcję rekurencyjną, należy po słowie `val` umieścić specjalne słowo kluczowe `rec`:

```
- val rec silnia = fn n =>
=      if n <= 0
=         then 1
=         else n * silnia (n-1);
val silnia = fn : int -> int
- silnia 5;
val it = 120 : int
```

Ponieważ funkcje definiuje się bardzo często, wprowadzono skróconą formę deklaracji funkcji w postaci:

$$\texttt{fun}\ f\ x\ \texttt{=}\ E$$

deklarującą funkcję f, która argumentowi x przyporządkowuje wartość E. Jest to skrócona forma deklaracji:

$$\texttt{val rec}\ f\ \texttt{=}\ \texttt{fn}\ x\ \texttt{=>}\ E$$

Na przykład funkcję `silnia` możemy zdefiniować jako

```
fun silnia n =
       if n <= 0
          then 1
          else n * silnia (n-1)
```

Ogólniej, zamiast

$$\texttt{val rec}\ f\ \texttt{=}\ \texttt{fn}\ x_1\ \texttt{=>}\ \texttt{fn}\ x_2\ \texttt{=>}\ \ldots\ \texttt{fn}\ x_n\ \texttt{=>}\ E$$

można napisać krócej

$$\texttt{fun}\ f\ x_1\ x_2\ \ldots\ x_n\ \texttt{=}\ E$$

np. obie poniższe deklaracje są równoważne:

```
val rec f = fn n => fn m =>
       if n = 0
          then m
          else f (n-1) (2*m)
```

oraz

```
fun f n m =
       if n = 0
          then m
          else f (n-1) (2*m)
```

2.6. Konteksty lokalne

Często wygodnie jest zdefiniować pewne wartości jedynie lokalnie, na użytek jednego wyrażenia lub kilku innych deklaracji (podobnie jak np. w blokach w C), tak by ich nazwy nie były widoczne globalnie w całym programie. Do tego celu służą w SML-u specjalne konstrukcje składniowe, zwane *kontekstami lokalnymi*: *wyrażenie* `let` i *deklaracja* `local`. *Wyrażenie* `let` ma postać:

`let` *deklaracje* `in` *E* `end`

gdzie *E* jest wyrażeniem, a *deklaracje* to m.in. nadanie nazwy, np.

```
- let
=    val x = 5
= in
=    27 * x
= end;
val it = 135 : int
- x;
Error: unbound variable or constructor: x
```

Identyfikator `x` jest związany z liczbą `5` jedynie od miejsca wystąpienia jego deklaracji do słowa `end` zamykającego wyrażenie.

Wyrażenia `let` używa się często, gdy zachodzi potrzeba wyliczenia pewnych wartości w treści funkcji, np. zamiast

```
fun f x y = (x+y)*(x+y)
```

lepiej napisać

```
fun f x y =
      let
         val z = x+y
      in
         z*z
      end
```

Jeśli chcemy, by *deklaracje-lokalne* były lokalne dla zestawu *deklaracji-globalnych*, wówczas posługujemy się *deklaracją* `local` postaci

```
local
   deklaracje-lokalne
in
   deklaracje-globalne
end
```

Na przykład

```
- local
=    val x = 5
= in
=    val y = x*x
= end;
val y = 25 : int
- y;
val it = 25 : int
- x;
Error: unbound variable or constructor: x
```

Tutaj podobnie, związano nazwę `x` z wartością `5` i `y` z wartością `25`. Widoczność identyfikatora `x` jest ograniczona jedynie do słowa `end` zamykającego deklarację lokalną.

2.7. Wzorce

Niektóre napisy (przeważnie identyfikatory i literały), zwane *konstruktorami*, służą do „konstruowania" wartości. Dla typów całkowitoliczbowych są to literały całkowite ze znakiem (np. `123`, `~23`, `0w5`) dla typu `char` to literały znakowe (np. `#"A"`), dla typu `string` to literały łańcuchowe (np. `"Ala"`), dla typu `unit` to `()` oraz `{}`, dla typu `bool` to stałe logiczne `true` i `false`, dla krotek — konstrukcja z nawiasami okrągłymi i przecinkami, dla rekordów — konstrukcja z nawiasami klamrowymi i przecinkami, dla list wreszcie — `nil` i `::` oraz konstrukcja z nawiasami kwadratowymi i przecinkami. Typ `real` nie posiada konstruktorów (ponieważ nie zdefiniowano dla niego relacji równości, por. także podrozdział 9.3.3).

Wzorzec w SML-u (ang. *pattern*) to wyrażenie złożone jedynie z konstruktorów i zmiennych, np. `(1,x)` jest wzorcem zawierającym zmienną `x`. Zmiennymi mogą być dowolne identyfikatory. Zwróćmy uwagę, że kontekst rozstrzyga o tym, czy dany identyfikator jest konstruktorem, czy zmienną.[8] Zmienne mogą być jedynie argumentami, nie mogą być funkcjami we wzorcu. Np. jeśli `f` nie jest konstruktorem (a zatem jest zmienną), to wyrażenie `f nil` nie jest poprawnym wzorcem. Wzorcem nie jest też np. `x@y`, ponieważ standardowa funkcja `@` łączenia list nie jest konstruktorem. Jeśli konstruktor nazwiemy w deklaracji `val`, nowa nazwa nie jest konstruktorem, np.:

[8] Jest to dosyć niebezpieczna i krytykowana cecha SML-a. Sytuację komplikuje fakt, że programista może wprowadzać nowe typy i nowe konstruktory. Łatwo zapamiętać, że `nil` jest konstruktorem, ale dla konstruktorów zdefiniowanych w programie jest to czasem kłopotliwe.

```
val Nil = nil
val Cons = op::
```

(słowo kluczowe `op` jest opisane w podrozdziale 3.9.1). Identyfikator `nil` jest konstruktorem listy pustej, wartością `Nil` jest również lista pusta, ale `Nil` nie jest konstruktorem, we wzorcu wystąpi więc jako zmienna. Podobnie `::` jest konstruktorem, zaś `Cons` zwykłym identyfikatorem i wzorzec `Cons(x,y)` jest błędny. Wyrażenie `Cons(x,y)` może być oczywiście użyte w programie, jednak nie jako wzorzec. Bycie konstruktorem jest zatem własnością syntaktyczną, przysługującą niektórym identyfikatorom, nie zaś własnością semantyczną, przysługującą wartościom.

Wzorzec *p dopasowuje się* do wartości v, jeśli za zmienne we wzorcu można podstawić takie wyrażenia, by wartością *p* po podstawieniu było v. Np.[9]

wzorzec	wartość	dopasowuje się?
`(x,1)`	`(true,1)`	tak, podstawienie: [`x`/`true`]
`[1,x,3]`	`[1,13,3]`	tak, podstawienie: [`x`/`13`]
`x::xs`	`[1,2,3]`	tak, podstawienie: [`x`/`1`, `xs`/`[2,3]`]
`x::xs`	`nil`	nie
`[x]`	`[1,2,3]`	nie

W SML-u argumentem formalnym funkcji może być nie tylko pojedyncza zmienna, ale także wzorzec. Podczas wywołania takiej funkcji najpierw argument faktyczny (który jest wartością) jest dopasowywany do wzorca, zmienne we wzorcu zostają związane z odpowiednimi wartościami i następnie obliczana jest wartość funkcji. Dla przykładu:

```
fun f (x,y) = x + y + 1
```

Podczas aplikacji `f` do argumentu (który musi być parą, żeby kontrola typów się powiodła), pierwszy element pary jest wiązany z nazwą `x`, a drugi — z `y`. Następnie obliczana jest wartość funkcji. Zatem w SML-u nie ma funkcji wielu zmiennych — każda funkcja ma jeden argument, który może być krotką. Funkcja

```
fun f x y = x + y + 1
```

też jest funkcją jednej zmiennej — jej parametrem jest `x`. W wyniku dostarcza ona funkcję (jednej zmiennej), której parametrem jest `y` i która w wyniku dostarcza sumę wartości zmiennych `x` i `y`.

Niestety nie zawsze argument dopasowuje się do wzorca. Dlatego zamiast jednej definicji możemy mieć ich szereg, oddzielonych znakami `|`:

`fun` *nazwa-funkcji* $wzorzec_1$ `=` $wyrażenie_1$
`|` *nazwa-funkcji* $wzorzec_2$ `=` $wyrażenie_2$
$\vdots$
`|` *nazwa-funkcji* $wzorzec_n$ `=` $wyrażenie_n$

[9] Napis $[x/e]$ oznacza *podstawienie* wyrażenia *e* w miejsce zmiennej x.

Dla przykładu:

```
fun len (x::xs) = 1 + len xs
  | len nil = 0
```

jest niezbyt efektywną, ale prostą funkcją wyznaczającą długość listy. Podczas obliczania wyniku aplikacji funkcji zdefiniowanej z użyciem wielu wzorców do argumentu, system próbuje dopasowywać argument faktyczny do wzorców w kolejności od pierwszego do ostatniego, wybiera pierwszy pasujący i oblicza wybraną gałąź definicji funkcji. Jeżeli żaden wzorzec nie dopasuje się do argumentu, zostanie zgłoszona sytuacja wyjątkowa.[10] Fragmenty definicji funkcji oddzielone znakami | nazywa się niekiedy *klauzulami* (ang. *clause*).

Funkcje należy definiować tak, by dla każdej możliwej wartości istniał w definicji funkcji co najmniej jeden wzorzec, do którego się ta wartość dopasuje. Mówimy wówczas, że w takiej definicji wzorce są *wyczerpujące* (ang. *exhaustive*).

Przestrzenie danych, które dopasowują się do różnych wzorców w jednej definicji mogą częściowo na siebie „zachodzić", choć jest najbardziej elegancko, gdy wzorce są *rozłączne* (ang. *exclusive*). Nie powinno jednak być wzorców, które nigdy nie będą miały szansy dopasować się do żadnej wartości, jak np. w poniższym przykładzie:

```
- fun f x = x
=   | f 0 = 1;
Warning: match redundant
           x => ...
    -->    0 => ...
val f = fn : int -> int
```

Pierwszy wzorzec `x` dopasowuje się do każdej wartości (ponieważ jest pojedynczą zmienną), zatem drugi wzorzec nigdy nie będzie wykorzystany. O takim wzorcu mówimy, że jest *nadmiarowy* (ang. *redundant*). Kompilator ostrzega przed takimi sytuacjami. Gdy wzorce nie są rozłączne, to ich kolejność może być ważna (por. ostatni przykład).

Wzorzec jest *liniowy* (ang. *linear*), jeśli każda zmienna występuje w nim nie więcej niż raz. Wszystkie wzorce w SML-u muszą być liniowe, dlatego np. poniższy program jest niepoprawny:

```
- fun f (x,x) = x
Error: duplicate variable in pattern(s): x
```

Jeżeli pewnej zmiennej we wzorcu nie wykorzystujemy w ciele funkcji, zamiast identyfikatora można napisać znak podkreślenia _, tj. *zmienną anonimową*, (ang. *wildcard*), np. w definicji funkcji

```
fun fst (x,_) = x
```

drugi element pary jest nieistotny (więc go nawet nie nazwano) i nie jest wykorzystywany w obliczeniach.

[10] Co w naszym przypadku jest równoznaczne z przerwaniem działania programu. W rozdziale 6.4 opisujemy jak reagować na takie sytuacje.

2.8. Kilka prostych przykładów

Używając rekursji i dopasowania wzorca można w Standard ML-u bardzo łatwo programować funkcje przetwarzające listy. Przyjrzyjmy się kilku przykładom. Funkcja `twice` podwaja wartość każdego elementu listy (który jest liczbą całkowitą), tj. `twice` $[x_1, \ldots, x_n] = [2{*}x_1, \ldots, 2{*}x_n]$:

```
fun twice (x::xs) = 2*x :: twice xs
  | twice nil = nil
```

Możemy ją zdefiniować jeszcze zwięźlej używając standardowej funkcji `map` (opisanej na stronie 8):

```
val twice = map (fn x => 2*x)
```

Aby móc obliczać sumę wszystkich elementów listy, wystarczy zdefiniować funkcję

```
fun sum (x::xs) = x + sum xs
  | sum nil = 0
```

Istotnie, `sum` $[x_1, \ldots, x_n] = x_1 + \ldots + x_n$. Korzystając ze standardowej funkcji `foldl` można ją też zdefiniować następująco:

```
val sum = foldl (fn (x,y) => x+y) 0
```

a nawet jeszcze krócej:

```
val sum = foldl op+ 0
```

(o słowie kluczowym `op` i operatorach w SML-u jest mowa w podrozdziale 3.9.1). Aby móc sprawdzać, czy dana lista ma parzystą długość, wystarczy napisać

```
fun evenLength (_::_::xs) = evenLength xs
  | evenLength [_] = false
  | evenLength [] = true
```

Rzeczywiście, lista pusta ma parzystą długość, lista jednoelementowa — nie, dłuższa lista ma parzystą długość, jeśli lista krótsza od niej o dwa elementy ma parzystą długość.

Czas na bardziej skomplikowany przykład. Chcemy policzyć „splot" dwu list, tj. listę na przemian elementów jednej i drugiej z podanych list:

$$\texttt{splice}\,[x_1, \ldots, x_n]\,[y_1, \ldots, y_m] = \begin{cases} [x_1, y_1, x_2, y_2, \ldots, x_n, y_n, y_{n+1}, \ldots, y_m], & \text{gdy } n \leq m \\ [x_1, y_1, x_2, y_2, \ldots, x_n, y_n, x_{n+1}, \ldots, x_n], & \text{gdy } n > m \end{cases}$$

Odpowiedni program w SML-u ma postać

```
fun splice (x::xs) (y::ys) = x :: y :: splice xs ys
  | splice nil ys = ys
  | splice xs nil = xs
```

Funkcje mogą być parametrami innych funkcji. Dla przykładu standardową funkcję `map` można by zdefiniować następująco:

```
fun map f (x::xs) = f x :: map f xs
  | map _ nil = nil
```

Na koniec zaprogramujemy algorytm *Quicksort* C.A.R. Hoare'a (zob. [6], str. 124–131) sortowania list liczb całkowitych (por. też podrozdział 18, w którym przedstawiono ogólniejszą i bardziej efektywną implementację tego algorytmu w SML-u). Idea algorytmu jest następująca: wybieramy jeden element listy (np. jej głowę) i nazywamy go *medianą*. Pozostałe elementy listy dzielimy na dwie podlisty elementów mniejszych i nie mniejszych od mediany. Te dwie listy sortujemy rekurencyjnie i na koniec spinamy razem, wstawiając pomiędzy nie medianę:

```
fun qsort (x::xs) : int list =
    let
       fun split (small,big) (y::ys) =
           if y<x
              then split (y::small,big) ys
              else split (small,y::big) ys
         | split (small,big) nil = qsort small @ [x] @ qsort big
    in
       split (nil,nil) xs
    end
  | qsort nil = nil
```

2.9. Więcej informacji o wzorcach

Szczególnie ciekawe są wzorce zawierające rekordy, np. funkcja

```
fun f {re=x,im=y} = x
```

bierze jako argument dwupolowy rekord, dopasowuje go do wzorca `{re=x,im=y}` zawierającego zmienne `x` oraz `y`, na skutek czego zmienna `x` zostaje związana z wartością pola `re` rekordu (`re` nie jest zmienną — jest etykietą pola rekordu, a więc częścią konstruktora), a `y` — z wartością pola `im` i na koniec zostaje obliczona wartość funkcji. Ponieważ etykiety pól rekordów i zmienne należą do rozłącznych klas, można do ich nazywania używać tych samych identyfikatorów:

```
fun f {re=re,im=im} = re
```

jest również poprawną wersją funkcji z poprzedniego przykładu. Wzorce takiej postaci można skracać, opuszczając znak równości i zmienne. Wówczas etykiety rekordu są jednocześnie nazwami zmiennych:

```
fun f {re,im} = re
```

Jeżeli w danej gałęzi nie korzystamy z wartości pewnych pól rekordu, możemy je pominąć, pisząc `...`, np.

```
fun f {re=0.0,im} = im
  | f {re,...} = re
```

Wzorzec zawierający wielokropek, to tzw. *elastyczny wzorzec rekordu*. Kompilator musi mieć jednak możliwość ustalenia typu wzorca, dlatego poniższa deklaracja jest niepoprawna:

```
- fun f {re,...} = re;
Error: unresolved flex record in let pattern
  type: {re:'Y, '...Z}
```

Można temu zaradzić np. jawnie podając typ argumentu (o typach mówimy w podrozdziale 9.1):

```
- fun f ({re,...} : {re:real,im:real}) = re;
val f = fn : {im:real, re:real} -> real
```

Jeżeli wartość M dopasowuje się do wzorca P, wówczas za zmienne w tym wzorcu są podstawiane odpowiednie „fragmenty" wartości M. Jeśli wzorzec P nie jest pojedynczą zmienną, wówczas nie ma zmiennej, z którą związana byłaby cała wartość M. Jeśli jednak np. w treści funkcji potrzebujemy wartości całego argumentu, musimy go „odbudować" z części, np.:

```
fun f (x::xs) = x + length (x::xs)
  | f nil = 0
```

Zmuszamy w ten sposób komputer do wykonania niepotrzebnej pracy. Dlatego w SML-u dowolny wzorzec P możemy zastąpić wzorcem postaci x `as` P, gdzie x jest zmienną. Wówczas po dopasowaniu wartości M do wzorca P, zostanie ona związana z identyfikatorem x, z którego można następnie korzystać, np.

```
fun f (l as x::_) = x + length l
  | f nil = 0
```

Wzorce ze słowem `as` można dowolnie zagłębiać w innych wzorcach. Ich użyteczność najlepiej pokazać na kilku przykładach:

`l as _::_`	znaczy niemal to samo, co pojedyncza zmienna `l`, ale dopasuje się jedynie do listy niepustej;
`x1::(xs as x2::_)`	dopasowuje się do listy co najmniej dwuelementowej, `x1` — do pierwszego elementu, `x2` — do drugiego, a `xs` — do ogona całej listy (zawierającego `x2` jako głowę);
`p as (x,y)`	`p` dopasuje się do całej pary, `x` — do pierwszego jej elementu, `y` — do drugiego;
`l as nil`	mało użyteczny, dopasuje się tylko do listy pustej i wówczas `l` zostanie związany z listą pustą.

Wzorce mogą pojawiać się także w abstrakcji funkcyjnej, np.

```
fn (0,_) => 1 | (_,y) => y
```

w nadaniu nazwy (tu może być tylko jeden wzorzec i nie może być wielu gałęzi), np.

```
val (x,_) = (1,2)
val (y::ys) = [1,2,3]
```

gdzie `x` i `y` otrzymują wartość `1`, a `ys` jest związany z listą `[2,3]`, oraz w specjalnym wyrażeniu, zwanym *wyrażeniem* `case` o następującej składni:

`case` *wartość* `of`
wzorzec$_1$ `=>` *wyrażenie*$_1$ `|`
wzorzec$_2$ `=>` *wyrażenie*$_2$ `|`
$\vdots$
wzorzec$_n$ `=>` *wyrażenie*$_n$

np.

```
fn x => (case x of nil => true | _  => false)
```

Ostatnie wyrażenie jest równoważne

```
fn nil => true | _ => false
```

Ponieważ wyrażenie `case` nie kończy się (niestety!) słowem kluczowym `end`, w gramatyce SML-a przyjęto, że `case` ma wyższy priorytet niż `fn`, zatem np.:

```
fn x => case x+1 of 0 => 1 | y => y
```

znaczy

```
fn x => (case x+1 of 0 => 1 | y => y)
```

Inny rozbiór gramatyczny można wymusić nawiasami:

```
fn x => (case x+1 of 0 => 1) | y => y
```

(zwróćmy uwagę, że jest to inna funkcja). Opuszczenie nawiasów w poniższym przykładzie spowoduje błąd składniowy, ponieważ kompilator „myśli”, że druga linijka jest kontynuacją wyrażenia `case`:

```
fun f (y::_) = (case y of 0 => 1 | y => y)
  | f nil = 0
```

Można je opuścić, gdy zmienimy kolejność wzorców:

```
fun f nil = 0
  | f (y::_) = case y of 0 => 1 | y => y
```

2.10. Podsumowanie

1. Predefiniowane typy proste, to `int`, `word`, `real`, `unit`, `char` i `string`.
2. Operatorem zmiany znaku jest `~` a nie `-`.
3. *Ścisła typizacja* wyklucza możliwość mieszania wartości różnych typów w jednym wyrażeniu. Należy używać jawnych funkcji konwersji nawet między danymi typów `int` i `real`.
4. Podstawowe operatory, takie jak `+` mogą być używane do danych różnych typów (są przeciążone).
5. *Rekordy* i *krotki* to kolekcje ustalonej liczby elementów dowolnych, być może różnych typów. *Listy* to kolekcje dowolnej liczby elementów tego samego typu.
6. Funkcje standardowe są dostępne w bibliotekach, odwołujemy się do nich podając nazwę struktury, w której są zdefiniowane, np. `Math.sin`.
7. Funkcje są takimi samymi wartościami, jak wartości typów prostych. Możemy tworzyć nowe funkcje używając abstrakcji funkcyjnej `fn` $\cdots$ `=>` $\cdots$.
8. Wartości można nazywać używając deklaracji `val` i odwoływać się do nich przez podanie ich nazwy.
9. Zasięg deklaracji pomocniczych należy ograniczać używając deklaracji `local` i wyrażenia `let`.
10. W SML-u do analizowania danych powszechnie używa się wzorców. Najlepiej, gdy wzorce wzajemnie się wykluczają i razem wyczerpują wszystkie możliwości.

2.11. Zadania

Zadanie 2.1. Poniżej opisano nieformalnie kilka funkcji przetwarzających listy. Zaprogramuj je w SML-u. Wartości niektórych z nich nie są określone dla wszystkich możliwych danych. W SML-u do obsługi takich sytuacji używa się *wyjątków* (por. rozdział 6.4). Jeśli nie umiesz się jeszcze posługiwać wyjątkami, zdefiniuj podane funkcje używając wzorców, które nie są wyczerpujące. Wówczas system SML-a sam zerwie obliczenia, jeśli te funkcje zostaną zaaplikowane do danych, dla których nie zostały zdefiniowane.

1. `hd` $[x_1, \ldots, x_n] = x_1$;
2. `tl` $[x_1, \ldots, x_n] = [x_2, \ldots, x_n]$;
3. `null`, dostarczająca wartości `true`, gdy jej argument jest listą pustą i `false` w przeciwnym razie;
4. `last` $[x_1, \ldots, x_n] = x_n$;
5. `member` x $[x_1, \ldots, x_n]$, dostarczająca wartości `true` gdy istnieje $i = 1, \ldots, n$, takie że $x = x_i$ i `false` w przeciwnym przypadku;

6. `exists` $p\ [x_1, \ldots, x_n]$, dostarczająca wartości `true`, gdy istnieje $i = 1, \ldots, n$, takie że $p\,x_i$ ma wartość `true` i `false` w przeciwnym razie;
7. `forall` $p\ [x_1, \ldots, x_n]$, dostarczająca wartości `true`, gdy $p\,x_i$ ma wartość `true` dla każdego $i = 1, \ldots, n$ i `false` w przeciwnym razie;
8. `find` $p\ [x_1, \ldots, x_n] = x_i$, gdzie i jest najmniejszym indeksem, dla którego $p\,x_i$ ma wartość `true`;
9. `removeOne` $p\ [x_1, \ldots, x_n] = [x_1, \ldots, x_{i_1}, x_{i+1}, \ldots, x_n]$, gdzie i jest najmniejszym indeksem, dla którego $p\,x_i$ ma wartość `true`;
10. `prefix` $p\ [x_1, \ldots, x_n] = [x_1, \ldots, x_{i-1}]$, gdzie i jest najmniejszym indeksem, dla którego $p\,x_i$ ma wartość `true`;
11. `suffix` $p\ [x_1, \ldots, x_n] = [x_{i+1}, \ldots, x_n]$, gdzie i jest najmniejszym indeksem, dla którego $p\,x_i$ ma wartość `true`;
12. `filter` $p\ [x_1, \ldots, x_n]$, dostarczająca listy tych elementów x_i spośród $x_1, \ldots, x_n$, dla których $p\,x_i$ ma wartość `true`;
13. `remove` $p\ [x_1, \ldots, x_n]$, dostarczająca listy tych elementów x_i spośród $x_1, \ldots, x_n$, dla których $p\,x_i$ ma wartość `false`;
14. `max` $[x_1, \ldots, x_n] = x_i$, jeśli x_i jest największym elementem na liście (liczb całkowitych) $[x_1, \ldots, x_n]$;
15. `zip` $([x_1, \ldots, x_n], [y_1, \ldots, y_n]) = [(x_1, y_1), \ldots, (x_n, y_n)]$;
16. `unzip` $[(x_1, y_1), \ldots, (x_n, y_n)] = ([x_1, \ldots, x_n], [y_1, \ldots, y_n])$;
17. `take` $k\ [x_1, \ldots, x_n] = [x_1, \ldots, x_i]$, gdzie $i = \min(\max(k, 0), n)$;
18. `drop` $k\ [x_1, \ldots, x_n] = [x_{i+1}, \ldots, x_n]$, gdzie $i = \min(\max(k, 0), n)$;
19. `flatten` $[x_1, \ldots, x_n] = x_1 \mathbin{@} \ldots \mathbin{@} x_n$;
20. `fromTo` $(m, n) = [m, m + 1, \ldots, n]$;
21. `genList` $f\ n = [f\,1,\ f\,2, \ldots,\ f\,n]$;
22. `nth` $i\ [x_1, \ldots, x_n] = x_i$;
23. `nthTail` $i\ [x_1, \ldots, x_n] = [x_{i+1}, \ldots, x_n])$;
24. `split` $i\ [x_1, \ldots, x_n] = ([x_1, \ldots, x_i], [x_{i+1}, \ldots, x_n])$;
25. `rmDupl` usuwającą z listy duplikaty, tj. wszystkie wystąpienia każdego elementu poza jego pierwszyn wystąpieniem;
26. `prod` wyznaczającą produkt dwu list, tj. listę wszystkich par elementów obu list:

$$\begin{aligned} \texttt{prod}\,([x_1, \ldots, x_m], [y_1, \ldots, y_n]) \quad = \quad [\quad & (x_1, y_1), \ldots, (x_1, y_n), \\ & (x_2, y_1), \ldots, (x_2, y_n), \\ & \qquad \vdots \\ & (x_m, y_1), \ldots, (x_m, y_n) \quad] \end{aligned}$$

Zadanie 2.2. Zaprogramuj funkcje `forall`, `filter`, `remove` i `flatten` z zadania 2.1 używając standardowych funkcji `foldl` i `foldr` i nie używając jawnie rekursji i dopasowania wzorca. Innymi słowy, wyraź podane funkcje za pomocą funkcji `foldl` i `foldr`. Z powodów, które są wyjaśnione w podrozdziale 11 funkcja `foldl` jest bardziej efektywna niż funkcja `foldr`. Tam, gdzie to możliwe, staraj się więc używać funkcji `foldl`.

Zadanie 2.3. *Podlistą* listy $l = [x_1, \ldots, x_n]$ nazywamy listę $[x_{i+1}, \ldots, x_j]$ dla $0 \leq i \leq j \leq n$. *Podciągiem* listy l nazywamy listę $[x_{i_1}, \ldots, x_{i_k}]$, gdzie $1 \leq i_1 < \ldots < i_k \leq n$, $k \geq 0$. *Permutacją* listy l nazywamy listę $[x_{\sigma(1)}, \ldots, x_{\sigma(n)}]$, gdzie $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ jest dowolną bijekcją. Napisz funkcje `sublist`, `subseq` i `perm` wyznaczające listy wszystkich podlist, podciągów i permutacji swoich argumentów. Kolejność podlist (podciągów, permutacji) na liście wynikowej jest nieistotna.

Zadanie 2.4. Aby móc przetwarzać liczby większe niż największa z liczb typu `int` (zwykle $2^{31}-1$) nieujemne liczby całkowite reprezentujemy jako listy liczb całkowitych z przedziału $0 \div 9$, poczynając od najmniej znaczącej, np. liczbę 146578934578343 reprezentujemy jako

```
[3,4,3,8,7,5,4,3,9,8,7,5,6,4,1]
```

Zaprogramuj algorytmy dodawania i mnożenia liczb w tej reprezentacji. Zaprogramuj funkcję `palindrom`, która dla zadanej liczby (w tej reprezentacji) wykonuje następującą czynność: dodaje do niej liczbę otrzymaną przez wypisanie cyfr w odwrotnej kolejności dopóty, dopóki wynik nie jest palindromem (przeczytany wprost i wspak daje tę samą liczbę), np. dla liczby 59 mamy: $59 + 95 = 154$, $154 + 451 = 605$, $605 + 506 = 1111$, zatem `palindrom 59` $=$ `1111`. Jaka jest największa wartość funkcji `palindrom` dla dwucyfrowego argumentu? Jaka jest wartość `palindrom 196` (obliczenia można przerwać klawiszem `ctrl-C`). Zaprogramuj funkcję, która wypisze do standardowego strumienia wyjściowego ładnie sformatowaną tabelkę wartości funkcji `palindrom` dla liczb z podanego przedziału.

Zadanie 2.5. Rozwiąż poprzednie zadanie w Pascalu lub C.

Zadanie 2.6. Zaprogramuj funkcję `search` p s obliczającą, ile razy napis p występuje (wystąpienia mogą się częściowo nakładać) w napisie s, np.

```
search "aba" "baabaababa"  =  3
```

Rozdział 3

Składnia języków programowania

3.1. Elementy teorii języków formalnych

3.1.1. Symbole, słowa i języki

Skończony niepusty zbiór Σ nazywamy *alfabetem*, a jego elementy *literami* lub *symbolami*. Alfabet jest *unarny*, jeśli zawiera tylko jedną literę, *binarny*, jeśli zawiera dwie litery. Skończony ciąg liter (alfabetu Σ) nazywamy *słowem* (*nad alfabetem* Σ). Zapis uw oznacza słowo (zwane *konkatenacją* słów u i w) powstałe przez wypisanie wszystkich liter słowa u, a następnie wszystkich liter słowa w. Ciąg długości 0 nazywamy *słowem pustym* i oznaczamy ϵ. Formalnie zbiór Σ^* słów nad alfabetem Σ definiujemy indukcyjnie:

$$\begin{aligned}
\Sigma^0 &= \{\epsilon\} \\
\Sigma^{k+1} &= \{wa \mid w \in \Sigma^k, a \in \Sigma\} \\
\Sigma^* &= \bigcup_{k=0}^{\infty} \Sigma^k \\
\Sigma^+ &= \bigcup_{k=1}^{\infty} \Sigma^k
\end{aligned}$$

Zbiór Σ^+ zawiera wszystkie *niepuste* słowa. *Językiem* (*nad alfabetem* Σ) nazywamy dowolny podzbiór zbioru Σ^*.

3.1.2. Gramatyki formalne

Gramatyki formalne wprowadzili Noam Chomsky i John Backus w latach 1956–58.

Gramatyką formalną (*gramatyką generacyjną*) nazywamy czwórkę $G = \langle \Sigma, V, S, P \rangle$, gdzie Σ i V są dwoma rozłącznymi alfabetami ($\Sigma \cap V = \emptyset$), Σ jest zwany alfabetem *terminalnym*, V zaś — *nieterminalnym*, S jest wyróżnionym symbolem nieterminalnym, zwa-

nym *symbolem startowym*, zaś P jest skończonym zbiorem *produkcji*, tj. par słów $u, w \in (\Sigma \cup V)^*$ zapisywanych w postaci $u \to w$, gdzie słowo u zawiera co najmniej jeden symbol nieterminalny. Słowo nad alfabetem terminalnym Σ nazywamy *słowem terminalnym*. Symbole nieterminalne bywają też nazywane *symbolami pomocniczymi* lub *kategoriami gramatycznymi*.

Na zbiorze słów nad alfabetem $\Sigma \cup V$ wprowadzamy binarną relację $\Rightarrow_G$, zwaną relacją *wyprowadzenia w jednym kroku* albo *relacją bezpośredniego wyprowadzenia* (*z gramatyki* G):

$$uxw \Rightarrow_G uyw \iff (x \to y) \in P; \quad u, w, x, y \in (\Sigma \cup V)^*$$

Relacja *wyprowadzenia* $\overset{*}{\Rightarrow}_G$ jest zwrotnym i przechodnim domknięciem relacji bezpośredniego wyprowadzenia, tj. jest najmniejszą zwrotną i przechodnią relacją binarną na słowach ze zbioru $(\Sigma \cup V)^*$ zawierającą relację $\Rightarrow_G$.

Wyprowadzeniem słowa w ze słowa u w gramatyce G nazywamy ciąg słów $u_0, \ldots, u_n \in (\Sigma \cup V)^*$, taki że $u_0 = u$, $u_n = w$, oraz $u_{i-1} \Rightarrow_G u_i$ dla $i = 1, \ldots, n$. *Wyprowadzeniem słowa w z gramatyki G* nazywamy jego wyprowadzenie z symbolu startowego S tej gramatyki.

Język *generowany przez gramatykę* G, oznaczany $\mathfrak{L}(G)$, jest zbiorem słów terminalnych wyprowadzalnych z symbolu startowego S:

$$\mathfrak{L}(G) \quad = \quad \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$$

3.1.3. Hierarchia Chomsky'ego

Wśród gramatyk formalnych Chomsky wyróżnił pewne klasy nakładając na postać produkcji gramatyki dodatkowe warunki:

Gramatyki struktur frazowych (typu 0): nie nakłada się żadnych warunków na postać gramatyki. Jeśli dla danego języka istnieje gramatyka struktur frazowych, która go generuje, język ten nazywa się *rekurencyjnie przeliczalnym*. Wszystkich języków (wszystkich podzbiorów Σ^*) jest kontinuum, gramatyk zaś przeliczalnie wiele. Istnieją zatem języki, które nie są rekurencyjnie przeliczalne.

Gramatyki kontekstowe (typu 1): każda produkcja ma postać

$$uAw \to uxw, \quad \text{gdzie } u, x, w \in (\Sigma \cup V)^*, A \in V, \text{ oraz } x \neq \epsilon$$

Zauważmy, że żaden język generowany przez taką gramatykę nie zawierałby słowa pustego. Dlatego dopuszczamy dodatkowo jeden wyjątek: gramatyka może zawierać produkcję

$$S \to \epsilon$$

ale wówczas S nie może wystąpić po prawej stronie żadnej produkcji tej gramatyki. Jeśli dla danego języka istnieje gramatyka kontekstowa, która go generuje, język ten nazywa się *językiem kontekstowym*.

Gramatyki bezkontekstowe (typu 2): każda produkcja gramatyki ma postać

$$A \to x, \quad \text{gdzie } A \in V, \text{ zaś } x \in (\Sigma \cup V)^*$$

Jeśli dla danego języka istnieje gramatyka bezkontekstowa, która go generuje, język ten nazywa się *językiem bezkontekstowym*.

Gramatyki regularne (typu 3): każda produkcja gramatyki ma postać

$$A \to xB \text{ lub } A \to x, \quad \text{gdzie } A, B \in V, \text{ zaś } x \in \Sigma^*$$

Jeśli dla danego języka istnieje gramatyka regularna, która go generuje, język ten nazywa się *językiem regularnym*.

Okazuje się, że hierarchia powyższa jest ściśle zstępująca, tzn. każdy język typu $i+1$ jest też językiem typu i. Dla każdego i istnieją jednak języki, które są typu i, lecz nie są typu $i+1$. Przykłady takich języków zawierają zadania 3.3 i 3.4. Dowody wspomnianych wyżej faktów wykraczają jednak poza ramy naszych rozważań. Więcej o językach i gramatykach można przeczytać w książce [73].

3.1.4. Gramatyki bezkontekstowe

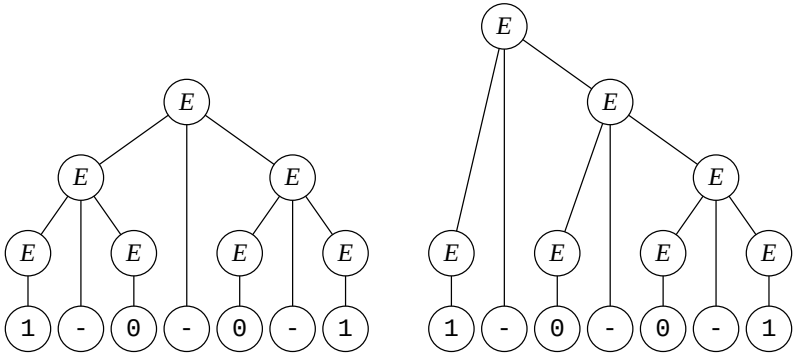
Produkcja $uAw \to uxw$ gramatyki kontekstowej mówi, że symbol nieterminalny A można przepisać do słowa x, ale pod warunkiem, że A występuje w odpowiednim kontekście, tj. jest otoczone słowami u i w. W gramatykach bezkontekstowych każda produkcja ma postać $A \to x$. Zatem produkcje gramatyki bezkontekstowej nie uwzględniają kontekstu (stąd nazwa). Jest to, jak powiedzieliśmy wyżej, istotne ograniczenie: istnieją języki kontekstowe, które nie są bezkontekstowe. Większość języków, zarówno komputerowych, jak i naturalnych, nie jest bezkontekstowa.

Przykład 3.1. Język C nie jest bezkontekstowy. Istotnie, w języku C wymaga się, by w tym samym zasięgu nazw nie występowały dwie deklaracje tej samej zmiennej z różnymi typami, np. program

```
void main (void) {
   int x;
   char x;
   return 0;
}
```

jest błędny. Powyższa własność nie jest wyrażalna w gramatyce bezkontekstowej.

Podejmowano próby stosowania gramatyk uwzględniających kontekst do opisu języków programowania. Np. van Wijngaarden opracował tzw. gramatyki dwupoziomowe (zob. [39]) które użyto do opisu języka Algol 68, jednak opis ten był całkowicie nieczytelny (zob. [98]). Ponadto zbyt ogólne gramatyki, np. gramatyki struktur frazowych bez nakładania na nie



Rysunek 3.1. Dwa przykładowe drzewa wyprowadzenia słowa `1-0-0-1` z gramatyki G opisanej równaniem 3.1

dodatkowych warunków lub gramatyki dwupoziomowe są niepraktyczne również z tego powodu, że nie istnieje dla nich algorytm sprawdzający, czy podane słowo należy do języka opisanego gramatyką, czy też nie (ten problem jest *nierozstrzygalny*). Jeżeli słowo istotnie należy do języka, to można jedynie to potwierdzić. Nie sposób jednak wyszukiwać błędów składniowych, tj. badać, czy dane słowo *nie należy* do języka. Dlatego do opisu języków programowania wybrano rozwiązanie pośrednie: buduje się gramatykę bezkontekstową opisującą język będący nadzbiorem opisywanego języka programowania, a następnie formułuje się dodatkowe warunki, jakie musi spełniać słowo, by do opisywanego języka należeć.

3.1.5. Drzewa wyprowadzenia

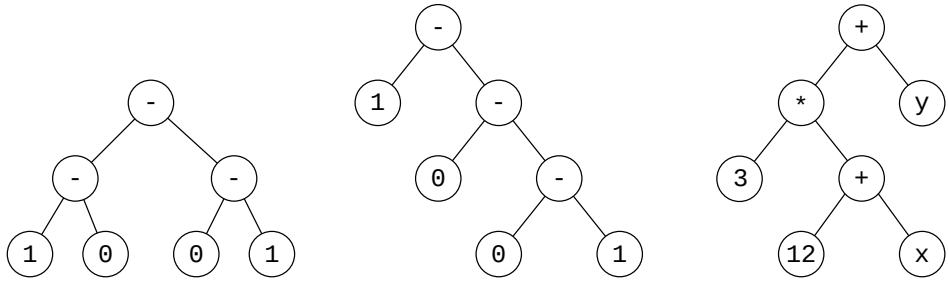
Drzewem wyprowadzenia (*rozbioru*) słowa $u \in \Sigma^*$ w gramatyce bezkontekstowej $G = \langle \Sigma, V, S, P \rangle$ nazywamy drzewo o wierzchołkach wewnętrznych etykietowanych symbolami nieterminalnymi i liściach etykietowanych symbolami terminalnymi lub symbolem ϵ, którego korzeń ma etykietę S, w którym dla każdego wierzchołka wewnętrznego, jeśli X jest jego etykietą, zaś $a_1, \ldots, a_k \in \Sigma \cup V$ — etykietami jego potomków w kolejności od lewego do prawego, to istnieje w P produkcja $X \to a_1 \cdots a_k$, oraz takie, że etykiety liści wypisane od lewej do prawej tworzą słowo u.
Dwa przykładowe drzewa wyprowadzenia słowa `1-0-0-1` z gramatyki

$$G \quad = \quad \langle \{0, 1, -\}, \{E\}, E, \{E \to E\text{-}E, E \to 0, E \to 1\} \rangle \tag{3.1}$$

są przedstawione na rysunku 3.1.

3.1.6. Jednoznaczność gramatyk

Gramatyka bezkontekstowa jest *jednoznaczna*, jeśli dla każdego słowa z języka przez nią generowanego istnieje dokładnie jedno drzewo wyprowadzenia tego słowa. W przeciwnym



Rysunek 3.2. Abstrakcyjne drzewa rozbioru wyrażeń `(1-0)-(0-1)`, `1-(0-(0-1))` oraz `3*(12+x)+y`

razie gramatyka jest *niejednoznaczna*.

Gramatyka opisana równaniem 3.1 jest niejednoznaczna. Ponieważ znaczenie słów definiuje się w oparciu o ich składnię, niejednoznaczność gramatyki może prowadzić do niemożności ustalenia sensu podanego słowa. Jeśli w ostatnim przykładzie `0` i `1` będziemy interpretować jako liczby naturalne, zaś `-` jako odejmowanie, to wyrażenie z ostatniego przykładu zinterpretowane zgodnie z podanymi drzewami rozbioru ma wartość albo 2 albo 0. Dla niejednoznacznych gramatyk należy ustalić, które z drzew rozbioru ma być użyte do zdefiniowania znaczenia danego słowa.

Uwagi. Istnieją języki bezkontekstowe, których nie da się opisać jednoznaczną gramatyką bezkontekstową. Taki problem występuje również w przypadku języków naturalnych, które są niejednoznaczne w podobnym sensie. Zdanie „*Jean fait manger les enfants*” [97] ma dwa znaczenia, zależnie od tego, czy *les enfants* jest dopełnieniem bliższym, czy dalszym (tj. czy dzieci jadły, czy były jedzone). Tekst z tabliczki umieszczanej we wrocławskich tramwajach o treści „*Motorniczy prowadzi sprzedaż biletów w dni wolne od pracy za odliczoną gotówkę*” ma co najmniej trzy różne drzewa rozbioru, którym odpowiadają trzy różne znaczenia (w dni powszednie wydaje resztę, czy nie sprzedaje biletów w ogóle, a może ma dzień wolny za odliczoną gotówkę?) Autor skryptu zastanawia się też, jakie umiejętności można zdobyć podczas „*Kursu nurkowania w miłej atmosferze*” reklamowanego na jednym z wrocławskich basenów pływackich.

3.2. Operatory i wyrażenia

Podstawową konstrukcją składniową większości języków programowania są *wyrażenia*. *Stałe* i *zmienne* są *wyrażeniami atomowymi*. Do budowania bardziej skomplikowanych wyrażeń służą *operatory*. Z każdym operatorem jest związana jego *arność*, tj. liczba argumentów. Operatory jednoargumentowe nazywamy *unarnymi*, dwuargumentowe zaś *binarnymi*. *Wyrażeniem* jest:

- wyrażenie atomowe,
- para złożona z operatora f o arności $n > 0$ i ciągu n wyrażeń, zwanych jego *argumentami*; operator f nazywamy *operatorem głównym* w tym wyrażeniu.

Abstrakcyjne drzewo rozbioru wyrażenia, to drzewo o wierzchołkach wewnętrznych etykietowanych operatorami i liściach etykietowanych wyrażeniami atomowymi, przy czym wierzchołek etykietowany operatorem o arności *n* ma *n* potomków. Rysunek 3.2 na poprzedniej stronie przedstawia abstrakcyjne drzewa rozbioru trzech przykładowych wyrażeń.

3.2.1. Składnia wyrażeń

Wyrażenia zapisuje się zwykle w następujący sposób. Do reprezentowania stałych używa się *literałów*, tj. specjalnych napisów reprezentujących stałe. Niekiedy wiele literałów oznacza tą samą stałą, np. SML-owe lierały `10`, `010` i `0xa` oznaczają tą samą stałą całkowitą 10. Do reprezentowania zmiennych używa się *identyfikatorów*. Ustalone napisy reprezentują operatory. Wyrażenie złożone z operatora o arności *n* i ciągu *n* wyrażeń można zapisać w notacji:

prefiksowej (przedrostkowej): wypisując najpierw operator, następnie wszystkie jego argumenty. Można (lecz nie jest to konieczne) oddzielać argumenty przecinkami i ujmować je w nawiasy (zapewnia to jednoznaczność w sytuacji, gdy operator może mieć zmienną liczbę argumentów). Wyrażenie z rysunku 3.2 na poprzedniej stronie zapiszemy w postaci prefiksowej jako:

```
+ * 3 + 12 x y
```

Z dodanymi przecinkami i nawiasami wygląda ciut czytelniej:

```
+(*(3,+(12,x)),y)
```

Notacji prefiksowej używa się powszechnie w językach programowania np. przy wywoływaniu funkcji. Notacja ta bywa nazywana *notacją polską* ponieważ wymyślił ją polski logik Jan Łukasiewicz. Jej zaletą jest możliwość nieużywania nawiasów przy zachowaniu jednoznaczności wyrażenia, dlatego bywa też nazywana *notacją beznawiasową*.

postfiksowej (przyrostkowej): podobnej do prefiksowej, przy czym operator wypisuje się po, a nie przed argumentami. Wyrażenie z rysunku 3.2 na poprzedniej stronie zapiszemy w postaci postfiksowej jako

```
3 12 x + * y +
```

Notacja ta bywa nazywana *odwrotną notacją polską*. Ma te same zalety, co notacja prefiksowa, ponadto istnieje prosty algorytm obliczania wartości wyrażeń zapisanych w tej notacji. Zauważmy że wyrażenie w notacji postfiksowej *nie* jest wyrażeniem w notacji prefiksowej zapisanym od tyłu.

infiksowej (wrostkowej): (dotyczy jedynie operatorów binarnych) w której operator zapisuje się pomiędzy jego argumentami. Przykładem jest wyrażenie `3*(12+x)+y`. Wadą tej notacji jest niejednoznaczność (np. czy `7-5-4` oznacza `(7-5)-4` i ma wówczas wartość `-2`, czy też `7-(5-4)` i ma wówczas wartość `6`) i związana z tym konieczność używania nawiasów.

miksfiksowej (mieszanej): nazwa operatora jest podzielona na kilka części, jego argumenty są wypisywane pomiędzy nimi. Przykładem jest *operator warunkowy* w języku C postaci b `?` x `:` y i SML-u postaci `if` b `then` x `else` y.

Należy odróżnić wyrażenia (reprezentowane przez ich abstrakcyjne drzewa rozbioru) od ich zapisu. To samo wyrażenie może być zapisane w postaci np. postfiksowej lub infiksowej. W postaci infiksowej może też być zapisane na wiele sposobów, np. z użyciem większej lub mniejszej liczby nawiasów. W praktyce używa się często kilku konwencji zapisu jednocześnie, np. operatory `+`, `*` itp. są zapisywane infiksowo, inne operatory (w tym definiowane przez użytkownika) — prefiksowo.

3.2.2. Operatory infiksowe

Wyrażenie postaci $e_1 \oplus e_2 \otimes e_3$, gdzie $\oplus$ i $\otimes$ są operatorami binarnymi zapisanymi infiksowo, nie jest jednoznaczne, tj. nie wiadomo, czy oznacza $(e_1 \oplus e_2) \otimes e_3$, czy też $e_1 \oplus (e_2 \otimes e_3)$. Aby nadać sens takim wyrażeniom wprowadza się dodatkowe reguły ich rozbioru. Z każdym operatorem związujemy liczbę naturalną, zwaną jego *priorytetem*, oraz ustalamy jego *łączliwość* (*kierunek wiązania*), tj. ustalamy, czy *łączy w lewo*, *w prawo*, czy może *nie jest łączny* w ogóle. Mówimy, że operator o *wyższym* (większym) priorytecie *wiąże silniej*. Algorytm „odtwarzania" nawiasów w wyrażeniu jest następujący: zaczynamy od operatorów o najwyższym priorytecie, jeśli wiążą w lewo, to od lewej strony wyrażenia, jeśli w prawo, to od prawej. Znajdujemy najmniejsze poprawnie zbudowane wyrażenie zawierające dany operator i ujmujemy je w nawiasy („wiążemy" jego argumenty). Następnie powtarzamy czynność dla operatorów o coraz niższych priorytetach. Np. jeśli operator $\oplus$ ma priorytet 3 i łączy w prawo, zaś $\otimes$ ma priorytet 17 i łączy w lewo, to w wyrażeniu $1 \oplus 2 \oplus 3 \otimes 4 \otimes 5 \oplus 6$ zaczynamy od operatora $\otimes$ od lewej strony: $1 \oplus 2 \oplus (3 \otimes 4) \otimes 5 \oplus 6$, następnie $1 \oplus 2 \oplus ((3 \otimes 4) \otimes 5) \oplus 6$, potem dla operatora $\oplus$ od strony prawej: $1 \oplus 2 \oplus (((3 \otimes 4) \otimes 5) \oplus 6)$, na końcu $1 \oplus (2 \oplus (((3 \otimes 4) \otimes 5) \oplus 6))$. Powyższe reguły nie gwarantują jednoznaczności w przypadku, gdy w wyrażeniu $e_1 \oplus e_2 \otimes e_3$ operatory $\oplus$ i $\otimes$ mają ten sam priorytet, przy czym $\oplus$ wiąże w lewo, zaś $\otimes$ w prawo (lub odwrotnie). Można temu zaradzić albo zabraniając nadawania tego samego priorytetu operatorom wiążącym w różnych kierunkach (np. żądając, by priorytety operatorów wiążących w lewo były nieparzyste, zaś wiążących w prawo — parzyste), lub zakładając, że w takim przypadku wszystkie operatory wiążą np. w lewo. Dla niektórych operatorów możemy przyjąć, że w ogóle nie są łączne. Wówczas dwa takie operatory nie mogą wystąpić w jednym wyrażeniu nie rozdzielone jawnie napisanymi nawiasami.

3.2.3. Opis wyrażeń za pomocą gramatyki bezkontekstowej

Niech A będzie symbolem nieterminalnym opisującym (pewne) wyrażenia atomowe, zaś $\{\oplus_1, \ldots, \oplus_n\}$ zbiorem operatorów binarnych o ustalonych priorytetach i kierunkach łączności. Chcemy opisać za pomocą gramatyki bezkontekstowej język wyrażeń zbudowanych z wyrażeń atomowych, wymienionych operatorów binarnych i nawiasów. Następujące produkcje dla symbolu nieterminalnego W (wyrażenia) tworzą najprostszą gramatykę opisującą

taki zbiór wyrażeń:

$$\begin{array}{lcll} W & \to & A & \text{wyrażenie atomowe jest wyrażeniem} \\ W & \to & (W) & \text{wyrażenie ujęte w nawiasy jest wyrażeniem} \\ W & \to & W \oplus_1 W & \\ & \vdots & & \text{dwa wyrażenia rozdzielone operatorem binarnym są wyrażeniem} \\ W & \to & W \oplus_n W & \end{array}$$

Taka gramatyka jest niejednoznaczna. Chcielibyśmy na podstawie drzewa wyprowadzenia danego wyrażenia móc zbudować jego abstrakcyjne drzewo rozbioru. Dla przykładu chcielibyśmy, aby drzewom wyprowadzenia z rysunku 3.1 na stronie 30 odpowiadały dwa pierwsze abstrakcyjne drzewa rozbioru przedstawione na rysunku 3.2 na stronie 31. Aby zapewnić jednoznaczność, gramatykę musimy nieco skomplikować. Niech priorytety operatorów będą liczbami z przedziału $1, \ldots, k$. Zamiast jednego symbolu nieterminalnego będzie ich teraz $k+1$. Symbol W_1 będzie symbolem startowym naszej gramatyki, W_{k+1} zaś będzie opisywał język złożony z wyrażeń atomowych i wyrażeń ujętych w nawiasy:

$$\begin{array}{lcl} W_{k+1} & \to & A \\ W_{k+1} & \to & (W_1) \end{array}$$

Język wyprowadzony z symbolu W_i będzie zawierał wyrażenia atomowe, dowolne wyrażenia ujęte w nawiasy i te spośród wyrażeń złożonych, w których operator główny ma priorytet co najmniej i. Dodajemy więc dla każdego operatora $\oplus_i$ priorytetu j po jednej produkcji:

$$\begin{array}{lcll} W_j & \to & W_j \oplus_i W_{j+1}, & \text{gdy } \oplus_i \text{ wiąże w lewo lub} \\ W_j & \to & W_{j+1} \oplus_i W_j, & \text{gdy } \oplus_i \text{ wiąże w prawo} \end{array}$$

nadto k produkcji postaci

$$W_j \to W_{j+1}, \quad \text{dla } j = 1, \ldots, k$$

Powyższy zbiór produkcji zadaje jednoznaczną (z wyjątkiem przypadku, gdy istnieją dwa operatory o tym samym priorytecie i różnych kierunkach łączności) gramatykę z symbolem startowym W_1 opisującą wyrażenia (por. zadanie 3.9).

Przykład 3.2. Wyrażenia złożone z czterech podstawowych działań arytmetycznych, w których priorytety i łączność operatorów są takie, jak przyjęto w matematyce, opisujemy następującym zbiorem produkcji

$$\begin{array}{lcl} W_3 & \to & A \\ W_3 & \to & (W_1) \\ W_2 & \to & W_3 \\ W_2 & \to & W_2 \times W_3 \\ W_2 & \to & W_2 \ / \ W_3 \\ W_1 & \to & W_2 \\ W_1 & \to & W_1 + W_2 \\ W_1 & \to & W_1 - W_2 \end{array}$$

3.3. Notacje używane do opisu języków programowania

3.3.1. Notacja BNF

Notacja matematyczna nie jest wygodna do zapisu gramatyk języków programowania. Dlatego około roku 1958 John Backus wprowadził specjalną notację, zmodyfikowaną rok później przez Petera Naura (redaktora definicji języka Algol 60 [124], w której ta notacja została po raz pierwszy użyta). Od nazwisk autorów notacja nazywa się BNF (Backus-Naur Form). Symbole nieterminalne zapisujemy w niej używając nawiasów kątowych ⟨ i ⟩, pomiędzy którymi możemy wpisać dowolny ciąg znaków różnych od nawiasów kątowych. Co prawda używanie pojedynczych liter na oznaczenie symboli nieterminalnych w notacji matematycznej jest bardziej zwięzłe, jednak w gramatyce opisującej język programowania występuje kilkadziesiąt symboli nieterminalnych. Dłuższe nazwy ułatwiają zrozumienie gramatyki. Symbole terminalne zapisujemy w takiej postaci, w jakiej występują w opisywanym języku. Zamiast symbolu → piszemy ::= (w latach 60-tych używano powszechnie maszyn do pisania i nie można było łatwo wydrukować strzałki). Dodatkowym udogodnieniem notacji BNF w stosunku do notacji matematycznej jest możliwość łączenia wielu produkcji, zawierających po lewej stronie ten sam symbol nieterminalny za pomocą symbolu „|". Dla przykładu gramatykę z rysunku 3.1 na stronie 30 zapiszemy w notacji BNF następująco:

⟨wyrażenie⟩ ::= `0` | `1` | ⟨wyrażenie⟩ `-` ⟨wyrażenie⟩

zaś liczby z opcjonalną kropką dziesiętną opiszemy tak:

⟨cyfra⟩ ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨ciąg cyfr⟩ ::= ⟨cyfra⟩ | ⟨ciąg cyfr⟩⟨cyfra⟩
⟨znak⟩ ::= `+` | `-` | ⟨puste⟩
⟨puste⟩ ::=
⟨liczba⟩ ::= ⟨znak⟩⟨ciąg cyfr⟩ | ⟨znak⟩⟨ciąg cyfr⟩ `.` ⟨ciąg cyfr⟩

3.3.2. Notacja z opisu języka C

Rozszerzeń i wariantów notacji BNF jest bardzo wiele. Praktycznie każdy autor opisu języka programowania używa własnej notacji, mniej lub bardziej zbliżonej do oryginalnej BNF. Na przykład autorzy definicji języka C użyli w książkach [82, 83] następującej notacji. Symbole nieterminalne zapisuje się *kursywą*. Jeśli składają się z kilku słów, pisze się między nimi łącznik. Symbole terminalne pisze się `pismem maszynowym` (lub pismem bezszeryfowym). Zamiast „::=" pisze się „:". Wariantów nie oddziela się kreską pionową, tylko pisze w kolejnych wierszach. Jeśli warianty są pojedynczymi symbolami lub krótkimi napisami, po dwukropku pisze się „jeden z:", a wszystkie warianty umieszcza w następnym wierszu. Napis $_{opc}$ po symbolu oznacza, że jest on opcjonalny (nie musi wystąpić). Liczby z opcjonalną kropką dziesiętną opiszemy w tej notacji następująco:

cyfra: jeden z:
`0 1 2 3 4 5 6 7 8 9`
ciąg-cyfr:
cyfra
ciąg-cyfr cyfra
znak: jeden z:
`+ -`
liczba:
$znak_{opc}$ *ciąg-cyfr*
$znak_{opc}$ *ciąg-cyfr* . *ciąg-cyfr*

3.3.3. Notacja EBNF

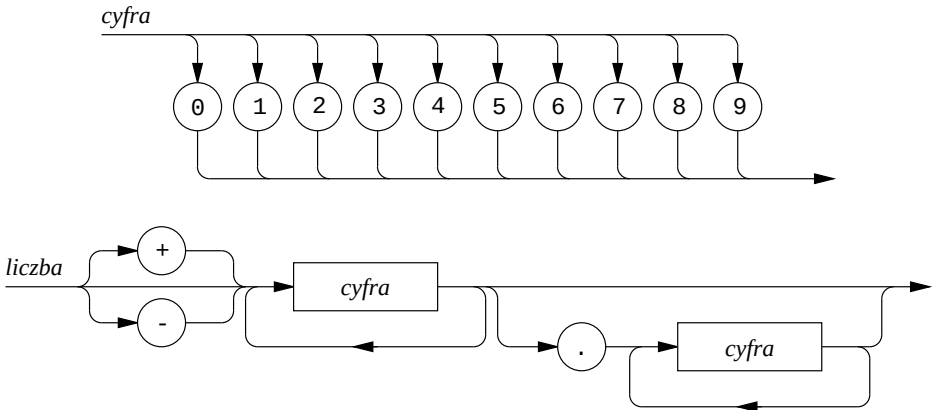
Notacja EBNF (*Extended BNF*) zawiera kilka rozszerzeń ułatwiających zwięzły zapis gramatyki. Symbole nieterminalne są w niej zawsze jednowyrazowe (jeśli nazwa symbolu nieterminalnego składa się z kilku wyrazów, piszemy między nimi łącznik) i nie ujmujemy ich w nawiasy kątowe ⟨ i ⟩. Symbole terminalne piszemy wyraźnie odmiennym krojem pisma (np. `maszynowym`), lub ujmujemy w cudzysłowy. Zamiast symbolu ::= używamy zwykłego znaku równości. Dodatkowo wprowadzamy trzy rodzaje nawiasów: okrągłe (i), które służą, tak jak w matematyce, do grupowania wyrażeń, kwadratowe [i], które oznaczają, że ujęty w nie napis jest opcjonalny, tj. może być pominięty przy wyprowadzaniu słowa i klamrowe { i }, które oznaczają dowolną liczbę (jedno lub więcej) powtórzeń danego ciągu. Zatem ujęcie pewnego ciągu symboli w nawiasy [{ }] odpowiada powtórzeniu zero lub więcej razy. Dla przykładu liczby z opcjonalną kropką dziesiętną można zwięźlej opisać gramatyką EBNF następująco:

$$\begin{aligned} \textit{cyfra} &= \text{"0"} \mid \text{"1"} \mid \text{"2"} \mid \text{"3"} \mid \text{"4"} \mid \text{"5"} \mid \text{"6"} \mid \text{"7"} \mid \text{"8"} \mid \text{"9"} \\ \textit{liczba} &= [\text{"+"} \mid \text{"−"}]\,\{\textit{cyfra}\}\,[\text{"."}\,\{\textit{cyfra}\}] \end{aligned}$$

Nawiasy w EBNF są jedynie wygodnym skrótem notacyjnym. Istotnie, możemy je wyeliminować z gramatyki kosztem zwiększenia liczby produkcji i symboli nieterminalnych. Jeżeli a jest ciągiem symboli terminalnych i nieterminalnych, a X i Y — symbolami nieterminalnymi nie występującymi w gramatyce, to:

- w miejsce napisu $[a]$ wstawiamy we wszystkich produkcjach symbol X i dodajemy produkcję $X = \epsilon \mid a$;
- w miejsce napisu $\{a\}$ wstawiamy we wszystkich produkcjach symbol X i dodajemy produkcje $Y = a$ oraz $X = Y \mid XY$;
- w miejsce napisu (a) wstawiamy we wszystkich produkcjach symbol X i dodajemy produkcję $X = a$.

Powyższe czynności powtarzamy aż do usunięcia wszystkich nawiasów z produkcji gramatyki (por. zadanie 3.15).



Rysunek 3.3. Diagram syntaktyczny

Wadą EBNF jest pewna trudność w ścisłym zdefiniowaniu pojęć drzewa wyprowadzenia i jednoznaczności gramatyki. Przez drzewo wyprowadzenia słowa z gramatyki EBNF będziemy zatem rozumieć drzewo wyprowadzenia tego słowa z gramatyki powstałej przez usunięcie nawiasów zgodnie z podanym wyżej schematem.

3.3.4. Diagramy syntaktyczne

Mówiąc o opisie języków programowania nie sposób nie wspomnieć o tzw. *diagramach syntaktycznych*. Są one mniej formalne od BNF, lecz bardziej czytelne. Dlatego pojawiają się częściej w podręcznikach programowania, rzadziej w formalnych definicjach języków. Przykład diagramu syntaktycznego dla liczb z opcjonalną kropką dziesiętną jest przedstawiony na rysunku 3.3.

3.3.5. Przykład opisu składni: wyrażenia arytmetyczne

Dla nabrania wprawy w posługiwaniu się gramatykami bezkontekstowymi, zbudujemy gramatykę opisującą wyrażenia złożone z liczb i czterech podstawowych działań arytmetycznych, których priorytety i kierunki łączności są takie, jak przyjęte w matematyce (por. przykład 3.2 na stronie 34).

Notacja BNF. Produkcje są wierną kopią gramatyki z przykładu 3.2. Zamiast symboli nieterminalnych W_1, W_2, W_3 i A wprowadzamy czytelniejsze nazwy ⟨wyrażenie⟩, ⟨składnik⟩, ⟨czynnik⟩ i ⟨liczba⟩:

$$\begin{aligned} \langle \text{cyfra} \rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \langle \text{liczba} \rangle &::= \langle \text{cyfra} \rangle \mid \langle \text{liczba} \rangle \, \langle \text{cyfra} \rangle \end{aligned}$$

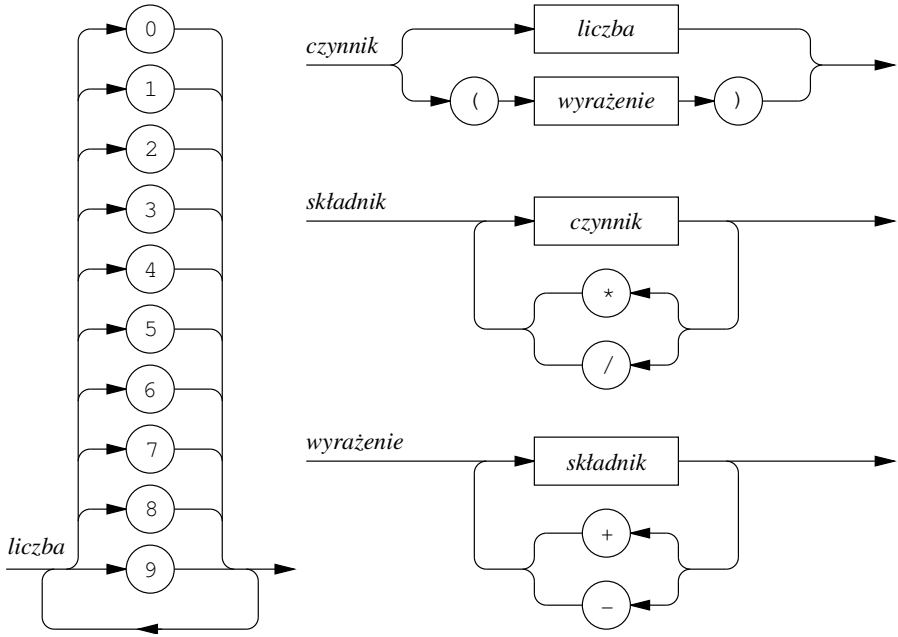
⟨czynnik⟩ ::= ⟨liczba⟩ | (⟨wyrażenie⟩)
⟨składnik⟩ ::= ⟨czynnik⟩
| ⟨składnik⟩ ⟨operator multiplikatywny⟩ ⟨czynnik⟩
⟨wyrażenie⟩ ::= ⟨składnik⟩
| ⟨wyrażenie⟩ ⟨operator multiplikatywny⟩ ⟨składnik⟩
⟨operator multiplikatywny⟩ ::= * | /
⟨operator addytywny⟩ ::= + | -

Wersja gramatyki z opisu języka C. Różnice w stosunku do BNF polegają głównie na innym sposobie zapisu.

cyfra: jeden z:
`0 1 2 3 4 5 6 7 8 9`
liczba:
cyfra
liczba cyfra
czynnik:
liczba
`(` *wyrażenie* `)`
składnik:
czynnik
składnik operator-multiplikatywny czynnik
wyrażenie:
składnik
wyrażenie operator-addytywny składnik
operator-multiplikatywny: jeden z:
`* /`
operator-addytywny: jeden z:
`+ -`

Notacja EBNF. Użycie nawiasów pozwala znacznie skrócić zapis w stosunku do BNF. Zauważmy, że możemy użyć nawiasów [{}] dzięki temu, że wszystkie operatory łączą w lewo. Gdyby łączyły w prawo, musielibyśmy, zgodnie z przyjętą przez nas definicją drzewa wyprowadzenia z gramatyki EBNF, jawnie użyć rekursji.

liczba = {"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"}
czynnik = *liczba* | "(" *wyrażenie* ")"
składnik = *czynnik* [{("*" | "/") *czynnik*}]
wyrażenie = *składnik* [{("+" | "−") *składnik*]}



Rysunek 3.4. Diagramy syntaktyczne opisujące wyrażenia

Diagramy syntaktyczne. Odpowiednie diagramy syntaktyczne są przedstawione na rysunku 3.4. Zastrzeżenia na temat jednoznaczności rozbioru słów zgodnie z gramatyką EBNF dotyczą także diagramów syntaktycznych.

3.4. Parser

Program sprawdzający, czy podany napis należy do języka opisanego ustaloną gramatyką bezkontekstową nazywamy *analizatorem składniowym* lub *parserem*. Zwykle parser nie tylko sprawdza poprawność składniową podanego tekstu, lecz w razie pomyślnego zakończenia analizy składniowej podejmuje również pewne *akcje semantyczne*, np. oblicza wartość przeczytanego wyrażenia, generuje kod wynikowy itp.

Program analizy składniowej można wyprowadzić wprost z gramatyki opisującej język, zapisanej np. w notacji EBNF. Do implementacji algorytmu użyjemy języka C. Przyjmijmy, że analizowany tekst jest przechowywany w pamięci w standardowej postaci, tj. w tablicy znaków i jest zakończony znakiem o kodzie `0`. Dla każdego symbolu nieterminalnego *X* zaprogramujemy funkcję o takiej samej nazwie:

```
bool X (char **napis, ...);
```

która w wyniku dostarczy odpowiedzi na pytanie, czy na początku `*napis`-u[1] znajduje się konstrukcja składniowa X.[2] Ponadto jeśli odpowiedź będzie pozytywna, funkcja `X` powinna zmodyfikować wartość `*napis` tak, by wskazywała na pierwszy znak po konstrukcji składniowej X. W razie negatywnej odpowiedzi funkcja `X` nie powinna modyfikować wartości `*napis`. Jeśli naszym celem będzie nie tylko analiza składniowa, ale także wykonanie pewnych czynności (np. obliczenie wartości wyrażenia, wygenerowanie kodu wynikowego itp.) funkcja `X` będzie posiadać dodatkowe parametry. Napisanie funkcji `X` dla produkcji

$$X \quad = \quad a$$

gdzie a jest pojedynczym znakiem jest bardzo łatwe:

```
bool X (char **napis) {
   if (**napis == 'a' ) {
      *napis++;
      return TRUE;
   } else return FALSE;
}
```

Dla produkcji postaci

$$X \quad = \quad Y$$

gdzie Y jest innym symbolem nieterminalnym, program będzie wyglądał następująco:

```
bool X (char **napis) {
   return Y(napis);
}
```

Zatem jeśli po prawej stronie produkcji jest symbol terminalny, należy sprawdzić, czy występuje on w napisie. Jeśli jest symbol nieterminalny, należy wywołać odpowiednią funkcję dla tego symbolu. Ciągowi symboli (terminalnych i nieterminalnych) będzie odpowiadać złożenie instrukcji w treści funkcji. Niestety nasz schemat nie będzie działał np. dla produkcji postaci

$$X \quad = \quad Xs$$

gdzie s jest dowolnym ciągiem symboli, tj. dla produkcji w których występuje tzw. *rekursja lewostronna*. Istotnie, wówczas funkcja `X` miałaby postać

```
bool X (char **napis) {
   if (X(napis))
      ...
```

[1] Napisy są wskaźnikami do znaków, są więc typu `char *`. Wskaźnik do napisu jest zatem typu `char **`.

[2] Ponieważ w C nie ma typu logicznego, należy go zdefiniować: `typedef enum {FALSE, TRUE} bool;` Tak jest czytelniej. Co prawda typ `bool` jest tożsamy z typem `int`, jednak informacja, że dana wartość jest traktowana jako wartość logiczna pomaga czytelnikowi zrozumieć program. A komputerowi jest wszystko jedno.

i próba jej wywołania zakończyłaby się natychmiast zapętleniem programu i przepełnieniem stosu. Na szczęście gramatykę (szczególnie EBNF) można zwykle łatwo przekształcić do postaci nie zawierającej lewostronnej rekursji. Rekursji w gramatyce będzie odpowiadać rekurencyjne wywoływanie funkcji. Warto więc w ogóle przebudować gramatykę usuwając rekursję (nie tylko lewostronną) tam, gdzie nie jest niezbędna (są w tym bardzo pomocne nawiasy { } i [{ }], dzięki którym rekursję można zastąpić iteracją).

Łatwo obmyślić schemat przekładu bardziej skomplikowanych produkcji. Alternatywie | będzie odpowiadać instrukcja wyboru lub instrukcja warunkowa, nawiasom kwadratowym [ ] — instrukcja warunkowa, nawiasom klamrowym { } — instrukcja `do` (powtórzenie jeden lub więcej razy), nawiasom [{ }] — instrukcja `while` (powtórzenie zero lub więcej razy).

Nie będziemy tutaj szczegółowo omawiać metod budowy takich parserów, odsyłając Czytelnika do książki Wirtha [190] lub dowolnego podręcznika teorii translacji, np. [180, 5]. W zamian podamy przykład parsera wyrażeń.

Rozważmy gramatykę EBNF opisującą wyrażenia przedstawioną na stronie 38. Podczas analizy składniowej pragniemy dodatkowo obliczyć wartość wyrażenia, jeśli jest ono poprawne (tj. zamierzamy zaprogramować prosty kalkulator). Dlatego wszystkie funkcje będą miały dodatkowy parametr `int *wynik`. Oto kompletny program:

```
/* kalkulator.c: parser wyrażeń obliczający ich wartość */

#include<stdio.h>
#include<ctype.h>

typedef enum {FALSE, TRUE} bool;

bool wyrazenie (char **napis, int *wynik);

bool liczba (char **napis, int *wynik) {
   char *pom = *napis;

   if (!isdigit(*pom)) return FALSE;
   *wynik = 0;
   do {
      *wynik = 10 * *wynik + *pom++ - '0';
   } while (isdigit(*pom));
   *napis = pom;
   return TRUE;
}

bool czynnik (char **napis, int *wynik) {
   char *pom = *napis;

   if (*pom == '(') {
      pom++;
```

```
      if (!wyrazenie(&pom,wynik)) return FALSE;
      if (*pom != ')') return FALSE;
      pom++;
   } else if (!liczba(&pom,wynik)) return FALSE;
   *napis = pom;
   return TRUE;
}

bool skladnik (char **napis, int *wynik) {
   char *pom = *napis, c;
   int n;

   if (!czynnik(&pom,wynik)) return FALSE;
   while (*pom == '*' || *pom == '/') {
      c = *pom++;
      if (!czynnik(&pom,&n)) return FALSE;
      if (c == '*')
         *wynik *= n;
      else *wynik /= n;
   }
   *napis = pom;
   return TRUE;
}

bool wyrazenie (char **napis, int *wynik) {
   char *pom = *napis, c;
   int n;

   if (!skladnik(&pom,wynik)) return FALSE;
   while (*pom == '+' || *pom == '-') {
      c = *pom++;
      if (!skladnik(&pom,&n)) return FALSE;
      if (c == '+')
         *wynik += n;
      else *wynik -= n;
   }
   *napis = pom;
   return TRUE;
}

bool oblicz (char *napis, int *wynik) {
   if (!wyrazenie(&napis,wynik)) return FALSE;
   if (*napis) return FALSE;
```

```
   return TRUE;
}

#define DLUGOSC_WIERSZA 80

int main (void) {
   char wiersz [DLUGOSC_WIERSZA];
   int wynik;

   printf("kalk> ");
   while (gets(wiersz) != NULL)
      if (oblicz(wiersz,&wynik))
         printf ("%d\nkalk> ", wynik);
      else printf ("Blad\nkalk> ");
   return 0;
}
```

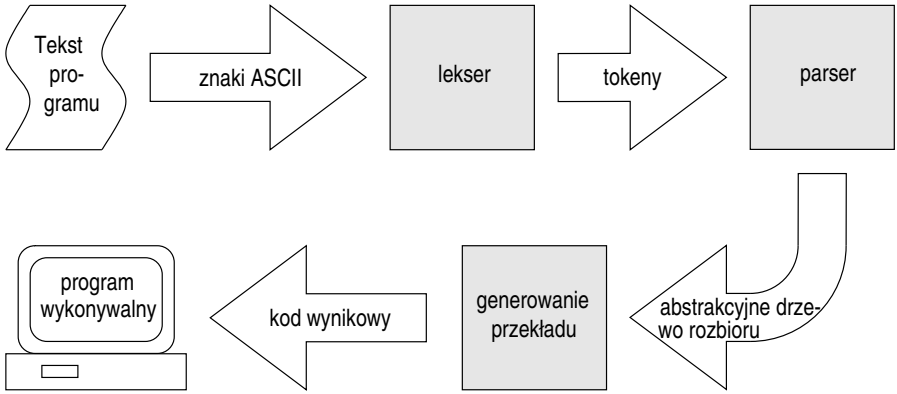
Funkcja `wyrazenie` sprawdza, czy `*napis` wskazuje na poprawne wyrażenie i jeśli tak, sprawia że `*napis` wskazuje na pierwszy znak po przeczytanym wyrażeniu, a `*wynik` zawiera obliczoną wartość wyrażenia. Pozostaje sprawdzić, czy za wyrażeniem nie znajdują się jakieś „śmieci”, tj. czy `*napis` wskazuje na znak o kodzie `0` (koniec napisu). Dokonuje tego funkcja `oblicz`.

3.5. Lekser

Nasz kalkulator z poprzedniego podrozdziału ma poważną wadę: pomiędzy wprowadzanymi znakami nie można umieszczać spacji. Nic dziwnego: nie uwzględniliśmy bowiem spacji w gramatyce. Niestety gramatyka (a więc i program z niej wyprowadzony) znacznie się skomplikuje, jeśli mielibyśmy dodać produkcję

$$\textit{spacje} \quad = \quad [\{\textvisiblespace\}]$$

gdzie $\textvisiblespace$ oznacza spację, i wstawiać symbol nieterminalny *spacje* wszędzie tam, gdzie mogą pojawić się spacje. Jest dużo wygodniej zmodyfikować procedurę wczytywania znaków i napisać funkcję, która będzie pomijać spacje przy kopiowaniu znaków ze strumienia wejściowego do tablicy. Możemy przy tym wykonać nieco więcej czynności, np. zamieniać ciągi cyfr na reprezentowane przez nie liczby, rozpoznawać operatory zapisywane za pomocą więcej niż jednej litery (np. `div`, `mod` itp.) Wówczas alfabetem terminalnym gramatyki bezkontekstowej nie będą pojedyncze znaki, lecz pewne abstrakcyjne obiekty, jak `LICZBA`, `OPERATOR_MNOZENIA` itp., niezależne od ich znakowej reprezentacji. Takie obiekty nazywamy *jednostkami leksykalnymi*, *leksemami* lub *tokenami*. Proces wczytywania tekstu rozpada się wówczas na dwie części: proces przekształcania znaków w tokeny, zwany *analizą leksykalną* i gramatyczny rozbiór ciągu tokenów, zwany *analizą składniową*. Program dokonujący analizy leksykalnej nazywa się *lekserem* lub *analizatorem leksykalnym*, program



Rysunek 3.5. Budowa kompilatora

analizy składniowej zaś *parserem* lub *analizatorem składniowym*. Tak jest zbudowany praktycznie każdy kompilator (por. rysunek 3.5) i ogólniej, każdy program przetwarzający teksty opisane nietrywialną gramatyką. Podział na analizę leksykalną i składniową ma również swoje odzwierciedlenie w opisie języków. Zwykle osobno opisuje się budowę tokenów języka, zaś w gramatyce używa się specjalnych symboli na ich oznaczenie. Z poziomu leksykalnego patrzymy na język jako na (dowolny) ciąg tokenów (bez wnikania w następstwo tokenów po sobie) i interesuje nas opis, z czego zbudowane są tokeny jako ciągi znaków. Składnia języka narzuca dalsze ograniczenia na postać poprawnych programów, tj. mówi jakie ciągi tokenów (bez wnikania w ich budowę znakową) tworzą poprawne programy. Semantyka języka, określając znaczenie programów, narzuca ostateczne warunki uznawania programów za poprawnie zbudowane (mówi np. że każdy identyfikator przed użyciem musi być zadeklarowany; gramatyka języka mówi tylko, że identyfikatory można deklarować i jak się to robi).

Przykład 3.3. Tokenami pewnego języka wyrażeń są: liczba, `(`, `)`, `+`, `-`, `*`, `div` i `mod`, przy czym liczba jest niepustym ciągiem cyfr `0` ÷ `9`. Umawiamy się, że pomiędzy dowolnymi dwoma tokenami (ale nie w ich wnętrzu) może pojawić się dowolna liczba *białych znaków*, tj. spacji, znaków tabulacji, zmiany wiersza itp. Białe znaki nie wpływają na znaczenie wyrażenia, dlatego mówimy o nich tylko na poziomie leksykalnym. Na tym poziomie powinny być pominięte.[3] Ponadto dowolny ciąg znaków różnych od `}` otoczony znakami `{` i `}` traktujemy jak biały znak (tj. pozwalamy na wstawianie *komentarzy*).

Pojedynczymi symbolami terminalnymi gramatyki bezkontekstowej są teraz tokeny `(`, `)`, `+`, `-`, `*`, `div`, `mod` i liczba i na poziomie składni już ich nie opisujemy. Gramatyka wyrażeń

[3]Dla bardziej skomplikowanych języków, aby uniknąć niejednoznaczności trzeba by dodatkowo założyć, że tokeny `div` i `mod`, gdyby występowały obok siebie (być może gramatyka to wykluczy, nas to jednak w tym miejscu nie interesuje) musiałyby być rozdzielone co najmniej jedną spacją (gdyby np. w naszym języku występowały identyfikatory, chcielibyśmy umieć identyfikator `divmod` odróżnić od pary operatorów `div mod`).

wygląda teraz następująco:

$$\begin{aligned} \textit{czynnik} &= \text{liczba} \mid (\ \textit{wyrażenie}\ ) \\ \textit{składnik} &= \textit{czynnik}\ [\{(* \mid \texttt{div} \mid \texttt{mod})\ \textit{czynnik}\}] \\ \textit{wyrażenie} &= \textit{składnik}\ [\{(+ \mid -)\ \textit{składnik}\}] \end{aligned}$$

Token liczba zapisaliśmy innym krojem pisma, by odróżnić go od symboli nieterminalnych gramatyki.

Lekser dla tak prostego języka można łatwo napisać *ad hoc* (dla bardziej skomplikowanych języków lekser trzeba budować systematycznie według pewnych metod, o których jeszcze powiemy, podobnie jak to uczyniliśmy budując parser wyrażeń). Napiszemy go dla przykładu w języku C. Parser możemy wyprowadzić w taki sam sposób, jak poprzednio, teraz jednak zamiast analizy kolejnych znaków łańcucha lub wczytywania pojedynczych znaków za pomocą funkcji `getchar`, napiszemy specjalną funkcję `gettoken`, w pewnym sensie odpowiednik funkcji `getchar` z poziomu znaków, która będzie dostarczać kolejne tokeny do analizy składniowej. Wpierw jednak musimy zdefiniować tokeny:

```
typedef enum {LICZBA,
              LEWY_NAWIAS, PRAWY_NAWIAS,
              PLUS, MINUS, RAZY, DIV, MOD,
              KONIEC} rodzaj_tokenu;
```

Ponieważ poza faktem, że przeczytany ciąg znaków jest liczbą, potrzebujemy pamiętać jej wartość, typ `token` będzie zatem strukturą:

```
typedef struct {
   rodzaj_tokenu rodzaj;
   int wartosc;
} token;
```

Jeżeli `rodzaj` = `LICZBA`, to pole `wartosc` zawiera tę liczbę. Umawiamy się, że nie będziemy wykorzystywać pola `wartosc` dla tokenów innych niż `LICZBA`. Wartość `KONIEC` ma pomocnicze znaczenie: funkcja `gettoken` zwróci `KONIEC`, jeśli napotka koniec strumienia wejściowego. Wynikiem działania funkcji `gettoken` będzie wartość logiczna zdania „wczytywanie następnego tokenu zakończyło się niepomyślnie” (może zakończyć się niepomyślnie, jeśli napotkany ciąg znaków nie jest żadnym poprawnym tokenem).[4] Oto analizator leksykalny:

```
bool gettoken (token *tok) {
   int c;
```

[4]Pewnie bardziej intuicyjne byłoby „myślenie pozytywne”, tj. przyjęcie, że wartością funkcji jest `TRUE`, jeśli analiza kolejnego leksemu przebiegła pomyślnie, jednak typowy kontekst użycia funkcji `gettoken` to:
`if (gettoken(&tok)) blad();`
i wówczas zawsze trzeba by używać negacji. Typ `bool` definiujemy tak jak w przypisie na stronie 40:
`typedef enum {FALSE, TRUE} bool;`

Zmienna c będzie przechowywać wczytywane znaki. Na początku pomijamy białe znaki i komentarze. Szczęśliwie wśród standardowych plików nagłówkowych w C mamy do dyspozycji ctype.h, a w nim funkcję isspace, która sprawdza, czy jej argument jest białym znakiem. Jeżeli napotkamy znak początku komentarza {, ignorujemy wszystkie znaki aż do napotkania znaku }. Wówczas proces opuszczania spacji i kolejnych komentarzy zaczyna się od nowa:

```
while (isspace(c=getchar()));
while (c=='{') {
   while ((c=getchar()) != '}');
   while (isspace(c=getchar()));
}
```

Obecnie zmienna c zawiera pierwszy znak leksemu. Należy więc sprawdzić, czym on jest:

```
switch (c) {
   case EOF: tok->rodzaj=KONIEC; return FALSE;
   case '+': tok->rodzaj=PLUS; return FALSE;
   case '-': tok->rodzaj=MINUS; return FALSE;
   case '*': tok->rodzaj=RAZY; return FALSE;
   case 'd': if (getchar()=='i' && getchar()=='v') {
                tok->rodzaj=DIV; return FALSE;
             } else return TRUE;
   case 'm': if (getchar()=='o' && getchar()=='d') {
                tok->rodzaj=MOD; return FALSE;
             } else return TRUE;
   default : if (isdigit(c)) {
                tok->rodzaj=LICZBA;
                tok->wartosc = 0;
                do {
                   tok->wartosc = 10*tok->wartosc + c - '0';
                } while (isdigit(c=getchar()));
                ungetc(c,stdin);
                return FALSE;
             } else return TRUE;
}
}
```

Ponieważ mamy tylko dwa wieloznakowe operatory, fragment programu, który je rozpoznaje ograniczyliśmy do minimum. Dla bardziej skomplikowanych języków trzeba by zaprogramować bardziej wyszukaną procedurę, prawdopodobnie przy użyciu biblioteki opisanej w pliku nagłówkowym strings.h.

Ciąg cyfr (napis) jest przekształcany na liczbę następująco. Jeżeli zmienna znakowa c zawiera cyfrę, to *wartość*(c) = c - '0' jest jest liczbową wartością tej cyfry (znaki w C

są utożsamiane z ich kodami; definicja języka przewiduje, że w zestawie znaków cyfry zajmują spójny obszar i są uporządkowane rosnąco; zatem np. kod znaku `8` jest o 8 większy od kodu znaku `0`). Rozważmy napis $c_1 \ldots c_k c_{k+1}$, gdzie c_i są cyframi. Jeśli przeczytaliśmy już $c_1 \ldots c_k$ i obliczyliśmy, że ten ciąg reprezentuje liczbę n, to ciąg $c_1 \ldots c_k c_{k+1}$ reprezentuje liczbę $10 \times n + \mathit{wartość}(c_{k+1})$. Zauważmy, że wczytywanie liczby kończymy na pierwszym znaku nie należącym już do liczby, należy go więc na powrót „oddać" do strumienia wejściowego.

3.6. Notacja postfiksowa

3.6.1. Stos

Stos S jest strukturą danych przechowującą (teoretycznie) dowolną liczbę elementów ustalonego typu E. Na stos można wstawiać elementy i można je z niego zdejmować. W każdej chwili mamy bezpośredni dostęp jedynie do elementu wstawionego jako ostatni. O takiej strukturze mówi się, że jest strukturą o dostępie sekwencyjnym typu LIFO (*last in, first out*). Elementy wstawione wcześniej są niedostępne do momentu usunięcia ze stosu elementów wstawionych później. Stos S elementów typu E jest zatem abstrakcyjnym typem danych, na którym można wykonywać następujące operacje:

$\mathit{init} : S \to \mathit{void}$	zainicjowanie stosu (sprawia, że stos jest pusty)
$\mathit{push} : S \times E \to \mathit{void}$	wstawienie elementu na wierzchołek stosu
$\mathit{top} : S \to E$	ujawnienie wierzchołka stosu
$\mathit{pop} : S \to E$	ujawnienie wierzchołka stosu i usunięcie go ze stosu
$\mathit{rem} : S \to \mathit{void}$	usunięcie wierzchołka stosu
$\mathit{empty} : S \to \mathit{bool}$	sprawdzenie, czy stos jest pusty

Próba wykonania operacji *top*, *pop* lub *rem* na pustym stosie kończy się błędem.

Użycie stosu pozwala uwolnić wiele algorytmów od niepotrzebnych detali technicznych związanych z przechowywaniem pewnych kolekcji elementów i osiągnąć wyższy poziom abstrakcji ułatwiający projektowanie i implementację takich algorytmów. Szczegóły techniczne związane z implementacją funkcji obsługujących stos są ukryte w osobnym module.

3.6.2. Implementacja stosu liczb całkowitych w języku C

W języku C specyfikacje modułów bibliotecznych umieszczamy w *plikach nagłówkowych*. Napiszemy zatem plik `stack.h`, w którym umieścimy definicję typu `stack` i prototypy funkcji na nim działających. Z punktu widzenia programisty wystarczyłaby jedynie deklaracja, że w implementacji stosu znajdzie się odpowiednia definicja typu `stack`, podobnie jak w specyfikacji są umieszczone jedynie deklaracje, że w implementacji stosu znajdą się odpowiednie definicje funkcji (tj. są umieszczane jedynie ich prototypy a nie pełne definicje funkcji). Z zewnątrz chcemy się odwoływać do danych typu `stack` jedynie poprzez zestaw tych funkcji i nie interesuje nas, jaką konkretną implementację typu `stack` przyjmiemy.

```
/* stack.h: specyfikacja stosu liczb całkowitych */

#define STACK_SIZE 200

typedef struct {
   int top;
   int tbl [STACK_SIZE];
} stack;

void init (stack *s);
int top (stack *s, int *x);
int push (stack *s, int x);
int pop (stack *s, int *x);
int rem (stack *s);
int empty (stack *s);
```

Tablica 3.1. Specyfikacja stosu w języku C

Niestety w języku C musimy w specyfikacji ujawnić definicję typu `stack`. Umawiamy się jednak, że jest to informacja jedynie dla kompilatora.

Liczby odkładane na stos będziemy przechowywać w kolejnych elementach tablicy o pewnym, z góry ustalonym rozmiarze `STACK_SIZE`. Musimy dodatkowo pamiętać, gdzie znajduje się wierzchołek stosu. Najprościej przechowywać indeks pierwszego wolnego elementu tablicy. Stos będzie zatem strukturą:

```
typedef struct {
   int top; /* indeks pierwszego wolnego elementu w tablicy */
   int tbl [STACK_SIZE]; /* tablica przechowująca elementy stosu */
} stack;
```

Wywołanie funkcji `top`, `pop` i `rem` może się niekiedy zakończyć niepowodzeniem (nie można bowiem ujawnić i/lub usunąć wierzchołka stosu, który jest pusty). Dodatkowo w przyjętej przez nas implementacji istnieje niebezpieczeństwo, że wszystkie elementy tablicy zostaną zajęte i wówczas nie powiedzie się wykonanie funkcji `push` (to jest błąd innego rodzaju: nie jest związany z nieprawidłowym użyciem stosu przez algorytm korzystający z naszej implementacji, lecz wynika z technicznych ograniczeń tej implementacji, których w idealnej sytuacji nie powinno być wcale). Przyjmujemy zatem, że powyższe cztery funkcje jako swój wynik zwracają informację, czy podczas wywołania funkcji wystąpił błąd (tj. `0`, jeśli wywołanie zakończyło się pomyślnie i `1` w przeciwnym razie), zaś funkcje `top` i `pop` zamiast zwracać wierzchołek stosu jako wynik, otrzymują jako dodatkowy parametr wskaźnik do zmiennej całkowitej, której wartość mają zmodyfikować. Funkcja `init` powinna zainicjować pole `top` struktury `stack`. Plik nagłówkowy `stack.h` przedstawia tablica 3.1, a

```
/* stack.c: implementacja stosu liczb całkowitych */

#include"stack.h"

void init (stack *s) {
   s->top=0;
}

int top (stack *s, int *x) {
   if (s->top==0)
      return 1;
   *x = s->tbl[s->top-1];
   return 0;
}

int push (stack *s, int x) {
   if (s->top>=STACK_SIZE)
       return 1;
   s->tbl[s->top] = x;
   s->top++;
   return 0;
}

int pop (stack *s, int *x) {
   if (s->top==0)
      return 1;
   *x = s->tbl[--(s->top)];
   return 0;
}

int rem (stack *s) {
   if (s->top==0)
      return 1;
   s->top--;
   return 0;
}

int empty (stack *s) {
   return s->top==0;
}
```

Tablica 3.2. Implementacja stosu w języku C

```
 1: init(S)
 2: while ¬ koniec wyrażenia do
 3:    t ← wczytaj-następny-token
 4:    if t jest liczbą then
 5:       push(S, t)
 6:    else
 7:       k ← arność operatora t
 8:       if na stosie S jest mniej niż k elementów then
 9:          błąd: przerwij działanie!
10:       end if
11:       zdejmij ze stosu k liczb i zaaplikuj do nich operator t
12:       push(S, wynik poprzedniego kroku)
13:    end if
14: end while
15: if liczba elementów na stosie S ≠ 1 then
16:    błąd: przerwij działanie!
17: else
18:    wynik ← pop(S)
19: end if
```

Tablica 3.3. Obliczanie wyrażenia w notacji postfiksowej

implementację stosu tablica 3.2.

3.6.3. Obliczanie wyrażeń w notacji postfiksowej

Wyrażenia w zapisie postfiksowym są przydatne nie tylko dlatego, że nie wymagają używania nawiasów, lecz także dlatego, że łatwo wyznaczyć ich wartość korzystając ze stosu. Pragniemy zbudować algorytm, który wczyta wyrażenie w notacji postfiksowej zbudowane z liczb i operatorów o podanej arności i pozostawi na stosie jego wartość. Jeżeli wyrażenie jest pojedynczą liczbą, to wystarczy ją wstawić na stos. Rozważmy wyrażenie $e_1 \ldots e_k f$, gdzie f jest operatorem o arności k, zaś są $e_1, \ldots, e_k$ wyrażeniami postfiksowymi. Aby obliczyć wartość wyrażenia $e_1 \ldots e_k f$ możemy uruchomić nasz algorytm k-krotnie w celu obliczenia wyrażeń $e_1, \ldots, e_k$. Dla każdego e_i algorytm pozostawi na stosie jego wartość. Wystarczy zatem zdjąć ze stosu k liczb, zaaplikować do nich k-argumentową funkcję reprezentowaną przez operator f i odłożyć wynik na stos. Algorytm jest zatem następujący: czytamy kolejne tokeny (liczby i operatory) zapisu postfiksowego. Liczby odkładamy na stos. Po napotkaniu operatora o arności k zdejmujemy k liczb ze stosu (jeżeli jest tam mniej niż k liczb, to znaczy, że wyrażenie jest niepoprawne) i aplikujemy do nich podaną operację. Jej wynik odkładamy na stos. Po zakończeniu obliczeń na stosie powinna znajdować się dokładnie jedna liczba, tj. wynik obliczeń. Odpowiedni algorytm jest zamieszczony w tablicy 3.3.

Implementacja kalkulatora obliczającego wyrażenia postfiksowe. Napiszemy program w języku C obliczający wyrażenia zapisane w notacji postfiksowej. Dla uproszczenia zaimplementujemy jedynie cztery podstawowe działania arytmetyczne. Wszystkie operatory występujące w wyrażeniu będą zatem binarne. Do wczytywania wyrażeń użyjemy leksera podobnego do opisanego w podrozdziale 3.5. Kompletny program jest zamieszczony poniżej.

```
/* konp.c: kalkulator posfiksowy */

#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include"stack.h"

void blad (const char *komunikat) {
   printf ("Awaria programu: %s\n", komunikat);
   exit(1);
}

typedef enum {FALSE, TRUE} bool;

typedef enum {LICZBA,
              PLUS, MINUS, RAZY, PODZIEL, POTEGA,
              KONIEC} rodzaj_tokenu;

typedef struct {
   rodzaj_tokenu rodzaj;
   int wartosc;
} token;

bool gettoken (token *tok) {
   int c;

   while ((c=getchar()) == ' ');
   switch (c) {
      case '\n': tok->rodzaj=KONIEC;
                   ungetc('\n',stdin);
                      return FALSE;
      case '+':  tok->rodzaj=PLUS; return FALSE;
      case '-':  tok->rodzaj=MINUS; return FALSE;
      case '*':  tok->rodzaj=RAZY; return FALSE;
      case '/':  tok->rodzaj=PODZIEL; return FALSE;
      case '^':  tok->rodzaj=POTEGA; return FALSE;
      default :  if (isdigit(c)) {
```

```
                    tok->rodzaj=LICZBA;
                    tok->wartosc = 0;
                    do {
                       tok->wartosc = 10*tok->wartosc + c - '0';
                    } while (isdigit(c=getchar()));
                    ungetc(c,stdin);
                    return FALSE;
                 } else return TRUE;
   }
}

int potega (int podstawa, int wykladnik) {
   int wynik = 1;

   if (wykladnik<0)
      return 0;
   while (wykladnik-->0)
      wynik *= podstawa;
   return wynik;
}

bool oblicz (int *wynik) {
   static stack st;
   token tok;
   int n, m;

   init(&st);
   if (gettoken(&tok)) return TRUE;
   while (tok.rodzaj != KONIEC) {
      if (tok.rodzaj == LICZBA) {
         if (push (&st, tok.wartosc)) blad ("przepelnienie stosu");
      } else {
         if (pop (&st,&n)) return TRUE;
         if (pop (&st,&m)) return TRUE;
         switch (tok.rodzaj) {
            case PLUS:    push(&st, m+n); break;
            case MINUS:   push(&st, m-n); break;
            case RAZY:    push(&st, m*n); break;
            case PODZIEL: push(&st, m/n); break;
            case POTEGA:  push(&st, potega(m,n)); break;
            default:      blad ("niemozliwa sekwencja instrukcji");
         }
      }
```

```
      if (gettoken(&tok)) return TRUE;
   }
   return pop(&st,wynik) || !empty(&st);
}

int main (void) {
   int wynik, c;

   printf("konp> ");
   while ((c=getchar()) != EOF) {
      ungetc (c,stdin);
      if (oblicz(&wynik))
           printf ("Blad\nkonp> ");
      else printf ("%d\nkonp> ", wynik);
      while (getchar()!='\n');
   }
   printf("\n");
   return 0;
}
```

3.6.4. Test na poprawność wyrażenia w notacji postfiksowej

Zauważmy, że wyrażenie w notacji postfiksowej jest poprawne wtedy i tylko wtedy, gdy algorytm z tablicy 3.3 zakończy się pomyślnie. Błąd może wystąpić wówczas, gdy na stosie nie ma dostatecznej liczby argumentów, albo gdy po przeczytaniu całego wyrażenia wysokość stosu jest różna od 1. Możemy zatem bez obliczania wyrażenia sprawdzić, czy jest ono poprawne, wyliczając wysokość stosu po przeczytaniu każdego tokenu. Wystarczy w tym celu dysponować pojedynczą zmienną *bilans*, początkowo równą zero (ponieważ stos początkowo jest pusty). Po przeczytaniu liczby zwiększamy *bilans* o jeden, po przeczytaniu operatora o arności k zmniejszamy *bilans* o $k - 1$ (bo zdejmujemy ze stosu k liczb i wstawiamy jedną). Wyrażenie jest poprawne, jeżeli w trakcie jego czytania *bilans* jest stale dodatni i po przeczytaniu całego wyrażenia ma wartość 1.

3.6.5. Przekład wyrażeń z notacji infiksowej na postfiksową

Zamierzamy zaprojektować algorytm dokonujący przekładu wyrażeń złożonych z liczb, operatorów i nawiasów z notacji infiksowej na notację postfiksową. Dla uproszczenia przyjmijmy, że wszystkie operatory są binarne. Dla przykładu rozważmy trzy grupy operatorów:

operator	priorytet	kierunek łączności
+, −	najniższy	w lewo
∗, /	średni	w lewo
^	najwyższy	w prawo

Program przekładu nie wnika w semantykę wyrażeń, ale dla ustalenia uwagi możemy myśleć, że operatory +, −, ∗ i / reprezentują cztery podstawowe działania arytmetyczne, zaś ^ potęgowanie. Wówczas tak ustalone priorytety i kierunki łączności zgadzają się z powszechnie przyjętymi umowami.[5] Operator potęgowania powinien łączyć w prawo, ponieważ napis i^{j^k} oznacza $i^{(j^k)}$, a nie $(i^j)^k$, zatem $i\hat{}j\hat{}k$ oznacza $i\hat{}(j\hat{}k)$.

Rozważmy najpierw wyrażenia bez nawiasów i operatora potęgowania. Zatem wszystkie operatory łączą w lewo. Jeżeli wyrażenie jest pojedynczą liczbą, to wystarczy ją przepisać na wyjście. Dla bardziej skomplikowanych wyrażeń użyjemy stosu do przechowywania już wczytanych tokenów, które mają być wypisane później. Kiedy zorientujemy się, że przeczytaliśmy już całe wyrażenie, te tokeny zostaną przepisane na wyjście. Rozważmy wyrażenie $e_1 \oplus e_2$, gdzie e_1 i e_2 są pewnymi wyrażeniami. Przetłumaczenie go na notację postfiksową polega na przetłumaczeniu wyrażenia e_1, następnie e_2 i na koniec wypisaniu operatora $\oplus$. Użyjemy zatem stosu do przechowania operatora $\oplus$ na czas wypisania wyrażenia e_2. Zgodnie z regułami pierwszeństwa operatorów wyrażenie e_1 może zawierać jedynie operatory o *nie mniejszym* priorytecie, zaś e_2 o *wyższym* priorytecie niż priorytet operatora $\oplus$ (bo wiąże on w lewo). Załóżmy, że nasz algorytm uporał się już z wczytaniem wyrażenia e_1. Na stosie może znajdować się jeszcze kilka tokenów oczekujących na koniec wyrażenia e_1 (i być może również tokeny oczekujące na zakończenie większego wyrażenia, którego częścią jest $e_1 \oplus e_2$, np. operator $\otimes$, jeśli ma niższy priorytet niż $\oplus$, a analizujemy wyrażenie $e_0 \otimes e_1 \oplus e_2$; operator $\otimes$ zejdzie ze stosu dopiero po przetłumaczeniu wyrażenia e_2). Zauważmy, że sygnałem końca wyrażenia e_1 jest przeczytanie operatora o priorytecie nie większym niż priorytet wszystkich operatorów występujących w e_1. Zatem ze stosu musimy przepisać na wyjście wszystkie operatory o nie mniejszym priorytecie. Następnie wstawiamy operator $\oplus$ na stos i rozpoczynamy wypisywanie wyrażenia e_2. Ponieważ zawiera ono jedynie operatory o wyższym priorytecie niż priorytet operatora $\oplus$, ten operator nie będzie usunięty ze stosu aż do końca wyrażenia e_2. Możemy już naszkicować nasz algorytm:

```
1:  while ¬ koniec-wyrażenia-infiksowego do
2:      przeczytaj token-wejściowy
3:      if token-wejściowy jest liczbą then
4:          wypisz (token-wejściowy)
5:      else
6:          przepisz ze stosu wszystkie tokeny o nie mniejszym priorytecie
                niż priorytet tokenu-wejściowego
7:          wstaw token-wejściowy na stos
8:      end if
9:  end while
10: przepisz wszystkie symbole ze stosu na wyjście
```

Zauważmy, że nie musimy traktować liczb w wyjątkowy sposób, jeśli przyjmiemy, że mają priorytet wyższy niż wszystkie operatory: możemy je wówczas wstawiać na stos, gdyż następny operator natychmiast je ze stosu usunie. Natomiast one same mając wyższy priorytet

[5] Dla operatora potęgowania wybraliśmy znak ^, ponieważ przypomina ↑, a zapis $n \uparrow m$ „udaje" n^m, jeśli nie można używać górnych indeksów.

nie spowodują wypisania żadnych operatorów. Zatem nasz algorytm można uprościć, jeśli rozszerzymy pojęcie priorytetu na wszystkie tokeny, nie tylko operatory:

1: **while** ¬ koniec-wyrażenia-infiksowego **do**
2: przeczytaj *token-wejściowy*
3: przepisz ze stosu wszystkie operatory o nie mniejszym priorytecie
niż priorytet *tokenu-wejściowego*
4: wstaw *token-wejściowy* na stos
5: **end while**
6: przepisz wszystkie symbole ze stosu na wyjście

Rozważmy obecnie operator potęgowania. Niestety nasz algorytm przetłumaczy wyrażenie `1^2^3` jako `1 2 ^ 3 ^`, ponieważ drugi symbol `^` zmusi pierwszy symbol `^` do opuszczenia stosu (bo mają ten sam priorytet). Rozważmy zatem algorytm:

1: **while** ¬ koniec-wyrażenia-infiksowego **do**
2: przeczytaj *token-wejściowy*
3: przepisz ze stosu wszystkie operatory o większym priorytecie
niż priorytet *tokenu-wejściowego*
4: wstaw *token-wejściowy* na stos
5: **end while**
6: przepisz wszystkie symbole ze stosu na wyjście

Ostatni algorytm będzie traktował wszystkie operatory jako łączące w prawo i przetłumaczy poprawnie wyrażenie `1^2^3` na `1 2 3 ^ ^`. Pozostaje połączyć ostatnie dwa algorytmy w jeden:

1: **while** ¬ koniec-wyrażenia-infiksowego **do**
2: przeczytaj *token-wejściowy*
3: **if** *token-wejściowy* wiąże w lewo **then**
4: przepisz ze stosu wszystkie operatory o *nie mniejszym* priorytecie
niż priorytet *tokenu-wejściowego*
5: **else**
6: przepisz ze stosu wszystkie operatory o *większym* priorytecie
niż priorytet *tokenu-wejściowego*
7: **end if**
8: wstaw *token-wejściowy* na stos
9: **end while**
10: przepisz wszystkie symbole ze stosu na wyjście

Dla liczb możemy przyjąć, że wiążą w lewo, gdyż i tak dwie liczby nigdy nie spotkają się na stosie. Zamiast modyfikować algorytm, możemy zdefiniować dla każdego tokenu dwa priorytety: *wejściowy* i *stosowy*. Dla właśnie wczytanego tokenu przyjmiemy, że jego priorytetem jest *priorytet-wejściowy*, zaś priorytetem tokenu znajdującego się na stosie będzie *priorytet-stosowy*. Wystarczy teraz sprawić, by *priorytet-wejściowy* $<$ *priorytet-stosowy*, jeśli operator łączy w lewo a *priorytet-wejściowy* $>$ *priorytet-stosowy*, gdy łączy w prawo, by nasz oryginalny algorytm rozróżniał kierunki łączności operatorów:

```
1: while ¬ koniec-wyrażenia-infiksowego do
2:     przeczytaj token-wejściowy
3:     przepisz ze stosu na wyjście wszystkie wierzchołki-stosu, takie że
           priorytet-wejściowy (token-wejściowy) < priorytet-stosowy (wierzchołek-stosu)
4:     wstaw token-wejściowy na stos
5: end while
6: przepisz wszystkie symbole ze stosu na wyjście
```

Pozostaje rozważyć nawiasy. Okazuje się, że wystarczy zdefiniować dla nich odpowiednie priorytety, by ostatni algorytm poprawnie przetwarzał wyrażenia z nawiasami. Zauważmy, że po przeczytaniu nawiasu otwierającego należy „zamrozić" całą zawartość stosu do czasu przeczytania odpowiadającego mu nawiasu zamykającego. Po przeczytaniu nawiasu otwierającego algorytm powinien pracować tak, jakby właśnie zaczynał przetwarzanie nowego wyrażenia z pustym stosem. Możemy zatem wstawić na stos nawias otwierający, nadając mu najniższy *priorytet-stosowy*. Wówczas żaden operator nie będzie mógł go (a więc i wszystkiego co jest pod nim) usunąć ze stosu. Aby nawias otwierający nie zmodyfikował zawartości stosu, jego *priorytet-wejściowy* musi być natomiast największy (jeszcze większy niż liczb). Kiedy przeczytamy nawias zamykający, znaczy to, że dotarliśmy do końca wyrażenia zaczynającego się po odpowiadającym mu nawiasie otwierającym. Powinniśmy zatem wypisać ze stosu wszystkie tokeny aż do znalezienia nawiasu otwierającego, a następnie usunąć ten nawias ze stosu. W tym jedynym przypadku zamiast wypisywać zdjęty ze stosu token na wyjście, po prostu go usuniemy. *Priorytetu-stosowego* nawiasu zamykającego nie musimy ustalać, ponieważ nawias zamykający nigdy nie znajdzie się na stosie. Właściwie dobrane priorytety są przedstawione w tablicy 3.5, a kolejny szkic naszego algorytm wygląda następująco:

```
1: while ¬ koniec-wyrażenia-infiksowego do
2:     przeczytaj token-wejściowy
3:     przepisz ze stosu na wyjście wszystkie wierzchołki-stosu, takie że
           priorytet-wejściowy (token-wejściowy) < priorytet-stosowy (wierzchołek-stosu)
4:     if priorytet-wejściowy (token-wejściowy) ≠ priorytet-stosowy (wierzchołek-stosu)
       then
5:         wstaw token-wejściowy na stos
6:     else
7:         {analizowane tokeny są parą odpowiadających sobie nawiasów}
8:         usuń wierzchołek-stosu
9:     end if
10: end while
11: przepisz wszystkie symbole ze stosu na wyjście
```

Zauważmy, że jeżeli analizowane wyrażenie jest w całości ujęte w nawiasy, to po jego wczytaniu stos jest pusty, ponieważ nawias zamykający usuwa z niego wszystkie elementy, aż do nawiasu otwierającego, który był wstawiony jako pierwszy. Możemy zatem przed rozpoczęciem wczytywania wyrażenia wstawić na stos nawias otwierający i dopisać na koniec

token	*priorytet-wejściowy*	*priorytet-stosowy*	*ranga*
`+, -`	1	2	-1
`*, /`	3	4	-1
`^`	6	5	-1
liczby	7	8	1
`(`	9	0	0
`)`	0	–	0

Tablica 3.4. Priorytety i rangi operatorów w algorytmie przekładu

wyrażenia nawias zamykający, by przepisywanie symboli ze stosu na wyjście no końcu algorytmu (linia 11) nie było konieczne.

Nasz algorytm będzie poprawnie działał w sytuacji, gdy wczytywane wyrażenie infiksowe będzie poprawne. W razie, gdy wczytywane wyrażenie jest niepoprawne, powinien to wykryć. Możemy np. sprawdzić, czy wypisane wyrażenie w notacji postfiksowej jest poprawnie zbudowane. W tym celu z każdym wypisywanym tokenem związujemy wartość *ranga* równą 1 dla liczb i $1 - k$ dla operatorów arności k i w czasie wypisywania wyrażenia przeprowadzamy test opisany w podrozdziale 3.6.4, tj. używamy zmiennej *bilans*, początkowo równej zero, którą po wypisaniu każdego tokenu zwiększamy o *rangę* tego tokenu. W razie gdy *bilans* przestanie być dodatni, lub na koniec będzie miał wartość różną od 1, sygnalizujemy błąd.

Zauważmy, że źle zbudowane wyrażenie może zostać przepisane do poprawnie zbudowanego wyrażenia w notacji postfiksowej, np. napis `1 2 +` zostaje przetłumaczony na `1 2 +` i nasz algorytm, nawet z opisanym powyżej testem nie sygnalizuje błędu. W wyrażeniu w postaci infiksowej liczby powinny występować na przemian z operatorami i pierwsza powinna być liczba. Możemy zatem użyć dodatkowej zmiennej *przeplot*, początkowo równej zero, którą będziemy w czasie wczytywania każdego tokenu zwiększać o jego *rangę*.[6] Jeśli przeplot stanie się mniejszy od zera lub większy niż jeden, to znaczy, że przeczytaliśmy niepoprawne wyrażenie. *Rangę* nawiasów ustalamy na równą zero. Tak rozbudowana procedura wykrywa już *prawie* wszystkie błędy we wczytywanych wyrażeniach (jakich nie wykrywa?). Wykaz priorytetów i rang operatorów znajduje się w tablicy 3.4, zaś kompletny algorytm przekładu — w tablicy 3.5.

Implementacja algorytmu przekładu. Do implementacji algorytmu przekładu opisanego w poprzednim podrozdziale użyjemy stosów z podrozdziału 3.6.2. Do wczytywania tokenów użyjemy leksera `gettoken` podobnego do opisanego w podrozdziale 3.5. Ponieważ zaimplementowaliśmy stos liczb całkowitych, a tokeny są strukturami a nie liczbami, dokonamy nieznacznej modyfikacji algorytmu. Zauważyliśmy na początku budowy algorytmu, że liczby nie muszą być odkładane na stos, tylko od razu wypisywane na wyjście. W naszym algorytmie liczba co prawda jest wstawiana na stos, ale zostaje z niego usunięta natychmiast po

[6] Takie rozwiązanie jest poprawne jedynie w przypadku, gdy wszystkie operatory są binarne.

```
 1: dopisz „)” na koniec wyrażenia infiksowego
 2: init (stos)
 3: push (stos, „(”)
 4: bilans ← 0
 5: przeplot ← 0
 6: while ¬ koniec-wyrażenia-infiksowego do
 7:    token-wejściowy ← następny-token-wyrażenia-infiksowego
 8:    przeplot ← przeplot + ranga (token-wejściowy)
 9:    if przeplot ≠ 0 ∧ przeplot ≠ 1 then
10:       błąd: przerwij działanie!
11:    end if
12:    if empty (stos) then
13:       błąd: przerwij działanie!
14:    else
15:       wierzchołek-stosu ← top (stos)
16:    end if
17:    {usuń symbole o wyższym priorytecie ze stosu}
18:    while priorytet-wejściowy (token-wejściowy) <
                 priorytet-stosowy (wierzchołek-stosu) do
19:       bilans ← bilans + ranga (wierzchołek-stosu)
20:       if bilans < 1 then
21:          błąd: przerwij działanie!
22:       end if
23:       rem (stos)
24:       wypisz (wierzchołek-stosu)
25:       wierzchołek-stosu ← top (stos)
26:    end while
27:    if token-wejściowy = ')' ∧ wierzchołek-stosu = '(' then
28:       rem (stos)
29:    else
30:       push (stos, token-wejściowy)
31:    end if
32: end while
33: if ¬empty (stos) ∨ bilans ≠ 1 then
34:    błąd: przerwij działanie!
35: else
36:    wyrażenie zostało pomyślnie przepisane
37: end if
```

Tablica 3.5. Algorytm przekładu z notacji infiksowej na postfiksową

przeczytaniu następnego tokenu. Dlatego na stosie wystarczy przechowywać jedynie rodzaj tokenu, zaś wartość liczby w pomocniczej zmiennej `doWypisania`, która ją przechowa do czasu przeczytania następnego tokenu. Kompletny program jest zamieszczony poniżej.

```
/* onp.c: zamiana wyrażeń infiksowych na postfiksowe */

#include<stdio.h>
#include<ctype.h>
#include"stack.h"

typedef enum {FALSE, TRUE} bool;

typedef enum {LICZBA, LEWY_NAWIAS, PRAWY_NAWIAS,
              PLUS, MINUS, RAZY, PODZIEL, POTEGA,
              KONIEC} rodzaj_tokenu;

int prio_wej[]  = {7,9,0,1,1,3,3,6,0};
int prio_stos[] = {8,0,0,2,2,4,4,5,0};
int ranga[]     = {1,0,0,-1,-1,-1,-1,-1,0};

typedef struct {
   rodzaj_tokenu rodzaj;
   int wartosc;
} token;

bool gettoken (token *tok) {
   int c;

   while ((c=getchar()) == ' ' || c == '\t');
   switch (c) {
      case '\n': tok->rodzaj=KONIEC;
                 ungetc('\n',stdin);
                    return FALSE;
      case '(':  tok->rodzaj=LEWY_NAWIAS; return FALSE;
      case ')':  tok->rodzaj=PRAWY_NAWIAS; return FALSE;
      case '+':  tok->rodzaj=PLUS; return FALSE;
      case '-':  tok->rodzaj=MINUS; return FALSE;
      case '*':  tok->rodzaj=RAZY; return FALSE;
      case '/':  tok->rodzaj=PODZIEL; return FALSE;
      case '^':  tok->rodzaj=POTEGA; return FALSE;
      default :  if (isdigit(c)) {
                    tok->rodzaj=LICZBA;
                    tok->wartosc = 0;
                    do {
```

```
                    tok->wartosc = 10*tok->wartosc + c - '0';
                  } while (isdigit(c=getchar()));
                  ungetc(c,stdin);
                  return FALSE;
               } else return TRUE;
   }
}

void wypisz (token *doWypisania) {
   switch (doWypisania->rodzaj) {
      case PLUS:    printf ("+ "); return;
      case MINUS:   printf ("- "); return;
      case RAZY:    printf ("* "); return;
      case PODZIEL: printf ("/ "); return;
      case POTEGA:  printf ("^ "); return;
      case LICZBA:  printf ("%d ", doWypisania->wartosc); break;
      default:      printf ("?? "); return;
   }
}

typedef enum {W_PORZADKU,
              PRZEPELNIENIE_STOSU,
                  NIEPOPRAWNE_WYRAZENIE} stan;

stan przetworz (void) {
   stack st;
   token tok, doWypisania;
   bool dalej=TRUE;
   int bilans=0;
   int przeplot=0;
   rodzaj_tokenu symb_wierzch;

   init (&st);
   if (push (&st,LEWY_NAWIAS)) return PRZEPELNIENIE_STOSU;
   while (dalej) {
      if (gettoken(&tok)) return NIEPOPRAWNE_WYRAZENIE;
      if (tok.rodzaj==KONIEC) {
         tok.rodzaj=PRAWY_NAWIAS;
         dalej=FALSE;
      }
      przeplot += ranga[tok.rodzaj];
      if (przeplot!=0 && przeplot!=1)
            return NIEPOPRAWNE_WYRAZENIE;
```

```
      if (top(&st, (int *) &symb_wierzch))
            return NIEPOPRAWNE_WYRAZENIE;
      while (prio_wej[tok.rodzaj] < prio_stos[symb_wierzch]) {
         bilans += ranga[symb_wierzch];
         if (bilans<1) return NIEPOPRAWNE_WYRAZENIE;
         rem(&st);
         doWypisania.rodzaj=symb_wierzch;
         wypisz(&doWypisania);
         if (top(&st, (int*) &symb_wierzch))
                return NIEPOPRAWNE_WYRAZENIE;
      }
      if (tok.rodzaj == LICZBA) doWypisania.wartosc=tok.wartosc;
      if (tok.rodzaj==PRAWY_NAWIAS && symb_wierzch==LEWY_NAWIAS)
         rem(&st);
      else if (push(&st,tok.rodzaj)) return PRZEPELNIENIE_STOSU;
   }
   if (empty(&st) && bilans == 1)
      return W_PORZADKU;
   else return NIEPOPRAWNE_WYRAZENIE;
}

int main (void) {
   int c;

   printf("onp> ");
   while ((c=getchar()) != EOF) {
      ungetc (c,stdin);
      switch (przetworz()) {
         case W_PORZADKU: printf ("\nonp> "); break;
         case PRZEPELNIENIE_STOSU:
            printf ("...\nBłąd: przepełnienie stosu\nonp> ");
            break;
         case NIEPOPRAWNE_WYRAZENIE:
            printf ("...\nBłąd: niepoprawne wyrażenie\nonp> ");
            break;
      }
      while (getchar()!='\n');
   }
   printf("\n");
   return 0;
}
```

3.7. Języki regularne

Warto by mieć matematyczne narzędzia do opisu języków na poziomie leksykalnym, takie jak gramatyki bezkontekstowe na poziomie składniowym. Okazuje się, że języki tokenów (np. język literałów całkowitoliczbowych czy zmiennopozycyjnych itp.) są językami regularnymi. Zatem taką rolę mogłyby pełnić gramatyki regularne. W praktyce używa się jednak powszechnie dwóch innych formalizmów opisu języków regularnych: *automatów skończonych* i *wyrażeń regularnych*.

3.7.1. Deterministyczne automaty skończone

Definicja 3.4. *Deterministyczny automat skończony* to piątka $\mathcal{A} = \langle \Sigma, Q, q_0, F, \delta \rangle$, gdzie Σ jest alfabetem, Q jest skończonym *zbiorem stanów automatu*, $q_0 \in Q$ jest wyróżnionym *stanem początkowym*, $F \subseteq Q$ jest zbiorem *stanów końcowych*, zaś $\delta : Q \times \Sigma \to Q$ jest *funkcją przejścia*. Funkcję przejścia rozszerzamy na dowolne słowa:

$$\begin{aligned} \hat{\delta}(q, \epsilon) &= q, \quad q \in Q \\ \hat{\delta}(q, wa) &= \delta(\hat{\delta}(q, w), a), \quad q \in Q, w \in \Sigma^*, a \in \Sigma \end{aligned}$$

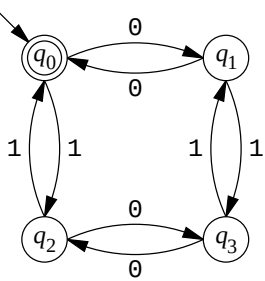
Słowo $w \in \Sigma^*$ jest *akceptowane* przez automat, jeśli $\hat{\delta}(q_0, w) \in F$. *Język akceptowany przez automat*, to zbiór akceptowanych przezeń słów:

$$\mathfrak{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

Ponieważ zarówno alfabet, jak i zbiór stanów automatu są skończone, funkcja przejścia δ jest określona na skończonej dziedzinie. Można ją zatem opisać za pomocą tabelki. Najwygodniej jednak przedstawić automat w postaci grafu skierowanego, którego wierzchołki są etykietowane stanami, krawędzie — literami alfabetu, tak że z każdego stanu wychodzi tyle krawędzi, ile liter ma alfabet i etykietowane są różnymi literami. Stany końcowe są oznaczone podwójnymi kółkami, dodatkowa strzałka oznacza stan początkowy q_0. Rysunek 3.6 na następnej stronie przedstawia automat akceptujący język złożony z tych ciągów zero-jedynkowych, w których liczba zer i liczba jedynek jest parzysta. Znaczenie stanów automatu jest następujące: automat jest po przeczytaniu słowa w w stanie q_0, jeśli $|w|_0$ i $|w|_1$ są parzyste ($|w|_0$ i $|w|_1$ oznaczają odpowiednio liczbę zer i jedynek w słowie w). Automat jest w stanie q_1, jeśli po przeczytaniu słowa w liczba $|w|_0$ jest nieparzysta, zaś $|w|_1$ — parzysta. Podobnie jest w stanie q_2 jeśli $|w|_0$ jest parzysta, zaś $|w|_1$ nieparzysta, oraz w stanie q_3, jeśli obie liczby $|w|_0$ i $|w|_1$ są nieparzyste.

Twierdzenie 3.5. Język jest regularny wtedy i tylko wtedy, gdy istnieje automat skończony, który go akceptuje.

Dowód tego twierdzenia można znaleźć w każdym podręczniku teorii języków formalnych, np. w [73], por. też zadanie 3.19.



Σ/Q	q_0	q_1	q_2	q_3
0	q_1	q_0	q_3	q_2
1	q_2	q_3	q_0	q_1

Rysunek 3.6. Automat skończony akceptujący język tych słów nad alfabetem {0, 1}, w których liczba zer i liczba jedynek jest parzysta

Σ/Q	q_0	q_1	q_2	q_3	q_4
INNY_ZNAK	q_4	q_4	q_4	q_4	q_4
SYMBOL_ZERO	q_1	q_0	q_3	q_2	q_4
SYMBOL_JEDEN	q_2	q_3	q_0	q_1	q_4

Tablica 3.6. Funkcja przejścia automatu z punktu 3.7.2

3.7.2. Implementacja automatów skończonych

Obecnie napiszemy program w języku C, który będzie ze standardowego strumienia wejściowego wczytywał ciąg znaków ASCII i sprawdzał, czy należy on do języka akceptowanego przez automat z poprzedniego paragrafu.

Alfabet ASCII liczy 256 znaków, z czego nas interesuje jedynie 0 i 1. Znaki ASCII będziemy więc odwzorowywać na trzy litery nowego automatu: INNY_ZNAK, SYMBOL_ZERO i SYMBOL_JEDEN, używając pomocniczej tablicy znak. Dzięki temu znacząco zmniejszymy rozmiar tablicy przejść automatu delta. Dodajemy dodatkowo jeden stan, do którego przejdzie automat po przeczytaniu znaku INNY_ZNAK. Stany będziemy kodować za pomocą liczb naturalnych. Funkcję przejścia automatu podajemy w tablicy 3.6. Zakodujemy ją w programie w tablicy delta. Oto program:

```
#include<stdio.h>
typedef enum {INNY_ZNAK, SYMBOL_ZERO, SYMBOL_JEDEN,
              LICZBA_SYMBOLI} znaki;
#define LICZBA_STANOW 5
```

Wartością identyfikatora LICZBA_SYMBOLI będzie zawsze aktualna liczba symboli, pod warunkiem jednak, że jest on wpisany w definicji typu znaki jako ostatni. Typ znaki zdefiniowaliśmy tak, by INNY_ZNAK miał wartość 0. Podanie zbyt krótkiego inicjatora tablicy wymusi wpisanie do wszystkich 256 elementów tablicy wartości 0, tj. właśnie INNY_ZNAK.

W treści funkcji zmienimy potem „ręcznie” dwa elementy tej tablicy. Tablicę `delta` przepisujemy wprost z tablicy 3.6 na poprzedniej stronie.

```
znaki znak[256] = {INNY_ZNAK};
const int delta [LICZBA_SYMBOLI] [LICZBA_STANOW] =
   {{4,4,4,4,4},
    {1,0,3,2,4},
    {2,3,0,1,4}};
const int koncowe [LICZBA_STANOW] = {1,0,0,0,0};
```

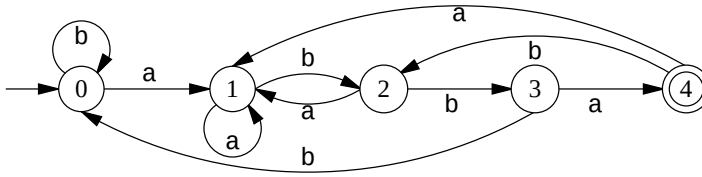
Funkcja `symuluj` wczytuje znaki ze standardowego strumienia wejściowego aż do napotkania znaku nowego wiersza i symuluje automat. Jej wynikiem jest odpowiedź na pytanie, czy przeczytane słowo należy do języka:

```
int symuluj (void) {
   int ch;
   int q;

   /* dokonczenie inicjowania tablicy znak */
   znak['0'] = SYMBOL_ZERO;
   znak['1'] = SYMBOL_JEDEN;

   q = 0; /* 0 jest stanem poczatkowym */
   while ((ch=getchar()) != '\n')
      q = delta[znak[ch]][q];
   return koncowe[q];
}
```

Program główny uruchamia funkcję `symuluj` i wypisuje wynik:

```
int main (void) {
   if (symuluj ())
        printf ("TAK\n");
   else printf ("NIE\n");
   return 0;
}
```

Taką implementację automatów spotyka się wszędzie tam, gdzie tekst źródłowy może być opisany za pomocą języka regularnego: w lekserze każdego kompilatora, w edytorach tekstu do wyszukiwania napisów itp. W praktyce automat ma również wyjście i w czasie przetwarzania tekstu generuje dane wyjściowe. Dla przykładu moglibyśmy napisać automat rozpoznający różne tokeny języka programowania. Przejście przez stan akceptujący będzie powodować przekazanie zanalizowanego tokenu do parsera. W wielu zastosowaniach automat nie jest zakodowany w programie na stałe, tylko wyliczany za pomocą odpowiedniego algorytmu.



Rysunek 3.7. Automat skończony wyszukujący wzorzec abba

3.7.3. Wyszukiwanie wzorca w tekście

Automat skończony można wykorzystać do szybkiego wyszukiwania ciągu znaków $w = w_1 \ldots w_m$ długości m w tekście x długości n. Zbudujemy automat $M = \langle \Sigma = \{\mathtt{a}, \mathtt{b}\}, Q = \{q_0, \ldots, q_m\}, q_0, F = \{q_m\}, \delta\rangle$, taki że po przeczytaniu pewnego słowa automat jest w stanie q_m wtedy i tylko wtedy, gdy słowo to jest postaci yw, dla pewnego słowa $y \in \Sigma^*$. Numer stanu automatu będziemy identyfikować z długością prefiksu wzorca w w następujący sposób: automat jest po przeczytaniu pewnego słowa w stanie q_i, jeśli ostatnio przeczytał i znaków wzorca w, tj. jeśli najdłuższy sufiks przeczytanego słowa będący jednocześnie prefiksem naszego wzorca ma długość i. Dla przykładu zbudujemy automat służący do wyszukiwania wzorca abba w tekście. Automat będzie miał pięć stanów. Załóżmy że w danej chwili znajduje się w stanie q_i. Zatem ostatnio przeczytał i pierwszych znaków wzorca, tj. słowo $w_1 \ldots w_i$ (być może wcześniej już coś czytał, ale nie pasowało to do wzorca). Co się stanie gdy przeczyta znak c? Jeśli $c = w_{i+1}$, to jasne, że przeczytał teraz $i + 1$ znaków wzorca, zatem przechodzi do stanu q_{i+1}. Ogólnie w słowie $w_1 \ldots w_i c$ musimy znaleźć najdłuższy sufiks, który jest jednocześnie prefiksem wzorca (są do tego szybkie algorytmy, nie będziemy jednak tu o nich mówić). Automat przejdzie do stanu, którego indeks jest długością tego sufiksu (jest to najdłuższe ostatnio przeczytane słowo, które rokuje nadzieję na uzupełnienie do pełnego wzorca). W naszym przypadku musimy zatem rozpatrzyć wszystkie prefiksy słowa abba. Dla prefiksu pustego, jeśli przeczytamy a, to słowo ϵa dalej jest prefiksem w i ma długość 1, przechodzimy więc do stanu q_1. Jeśli zaś przeczytamy b, to pozostajemy w stanie q_0. Podobnie analizujemy pozostałe stany. Dla przykładu jeśli jesteśmy w stanie q_2, to znaczy, że najdłuższy ostatnio przeczytany ciąg, który da się uzupełnić do wzorca (tj. jest jego prefiksem) ma długość 2 (i jest nim ab). Jeśli teraz przeczytamy a, to ostatnio przeczytaliśmy aba. Jednak wzorzec jest postaci abba, zatem najdłuższy ostatnio przeczytany ciąg, będący prefiksem wzorca, to a (możemy to sprawdzić w pętli: dla j od 1 do długości sprawdzanego ciągu — tu 3 — sprawdzamy czy jego sufiks długości j i prefiks wzorca długości j są równe; wybieramy największe j dla którego tak jest, lub 0, jeśli tak nie było dla żadnego j). Dlatego ze stanu q_2 po przeczytaniu a przechodzimy do q_1, po przeczytaniu zaś b — do q_3. Cały automat jest pokazany na rysunku 3.7. Po zbudowaniu automatu używamy procedury z podrozdziału 3.7.2 do jego symulacji. Przy każdym osiągnięciu stanu akceptującego program symulujący automat może np. wypisywać pozycję znalezionego wzorca w tekście, zastępować znaleziony wzorzec innym ciągiem znaków lub zliczać liczbę wystąpień. Zauważmy że przeszukanie tekstu długości n w poszukiwa-

niu ustalonego wzorca długości m (pomijając czas niezbędny na zbudowanie automatu) jest proporcjonalne do n (nie zależy od wielkości wzorca). Naiwne wyszukiwanie jest proporcjonalne do iloczynu $n \times m$. Automat można łatwo zbudować w czasie proporcjonalnym do m^3 (a istnieją algorytmy pozwalające go zbudować w czasie proporcjonalnym do m). Jeśli $n = 2^{20} \approx 1\,\mathrm{MB}$ a $m = 2^8 = 256\,\mathrm{B}$, to $m \times n = 2^{28}$, podczas gdy $m^3 + n \approx 2^{24}$, zaś $m + n \approx 2^{20}$. Można więc mieć nadzieję, że nasz program będzie działał dla takich danych od 16 do 256 razy szybciej niż naiwne przeszukiwanie (tj. np. 1 minutę zamiast 16 minut czy ponad 4 godzin). Zachęcam do sprawdzenia w praktyce!

3.7.4. Wyrażenia regularne

Wyrażenia regularne nad alfabetem $\Sigma = \{0, 1\}$ są następującej postaci:

1. 0, 1, ϵ i $\emptyset$ są wyrażeniami regularnymi;
2. jeśli e_1 i e_2 są wyrażeniami regularnymi, to są nimi także e_1^*, (e_1), $e_1 e_2$ i $e_1 + e_2$;
3. wszystkie wyrażenia regularne są postaci opisanej w punktach 1 i 2.

Zauważmy, że tak nieformalnie opisana składnia wyrażeń regularnych jest niejednoznaczna (nie wiemy, czy $0 + 1^*$ oznacza $(0 + 1)^*$, czy też $0 + (1^*)$). Dlatego musimy dodatkowo określić reguły rozbioru takich wyrażeń. Przyjmijmy, że * wiąże najsilniej, konkatenacja (napisanie dwóch wyrażeń jedno za drugim) słabiej i w lewo, $+$ zaś najsłabiej i również w lewo. Formalnie więc składnię wyrażeń regularnych możemy opisać jednoznaczną gramatyką bezkontekstową:

$$
\begin{array}{rcl}
\langle\text{wyrażenie proste}\rangle & ::= & 0 \mid 1 \mid \epsilon \mid \emptyset \mid (\langle\text{wyrażenie regularne}\rangle) \\
\langle\text{domknięcie Kleene'ego}\rangle & ::= & \langle\text{wyrażenie proste}\rangle \\
& \mid & \langle\text{domknięcie Kleene'ego}\rangle^* \\
\langle\text{konkatenacja}\rangle & ::= & \langle\text{domknięcie Kleene'ego}\rangle \\
& \mid & \langle\text{konkatenacja}\rangle\ \langle\text{domknięcie Kleene'ego}\rangle \\
\langle\text{wyrażenie regularne}\rangle & ::= & \langle\text{konkatenacja}\rangle \\
& \mid & \langle\text{wyrażenie regularne}\rangle + \langle\text{konkatenacja}\rangle
\end{array}
$$

Niech $\mathfrak{L}(e)$ oznacza język opisywany przez wyrażenie regularne e. Wtedy:

$$
\begin{aligned}
\mathfrak{L}(0) &= \{0\} \\
\mathfrak{L}(1) &= \{1\} \\
\mathfrak{L}(\epsilon) &= \{\epsilon\} \\
\mathfrak{L}(\emptyset) &= \emptyset \\
\mathfrak{L}(e^*) &= (\mathfrak{L}(e))^* \\
\mathfrak{L}(e_1 e_2) &= \{uw \mid u \in \mathfrak{L}(e_1), w \in \mathfrak{L}(e_2)\} \\
\mathfrak{L}(e_1 + e_2) &= \mathfrak{L}(e_1) \cup \mathfrak{L}(e_2)
\end{aligned}
$$

gdzie *domknięcie Kleene'ego* L^* dowolnego języka L definiujemy jako zbiór konkatenacji dowolnej liczby słów z języka L:

$$L^* = \{w_1 \dots w_k \mid w_1, \dots, w_k \in L, k \geq 0\}$$

Formalnie:

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^{k+1} &= \{uw \mid u \in L^k, w \in L\} \\ L^* &= \bigcup_{k=0}^{\infty} L^k \end{aligned}$$

Dla przykładu $(0+1)^*101(0+1)^*$ opisuje język tych słów, które zawierają podciąg `101`.

Twierdzenie 3.6. Język jest regularny wtedy i tylko wtedy, gdy istnieje wyrażenie regularne, które go opisuje.

Dowód tego twierdzenia można znaleźć w każdym podręczniku teorii języków formalnych, np. w [73]. Zatem mamy trzy narzędzia opisu języków regularnych: gramatyki regularne, automaty skończone i wyrażenia regularne.

3.7.5. Wyrażenia regularne w Unix-ie

Wyrażenia regularne opisują pewne języki. Jeśli napis należy do języka opisanego przez wyrażenie, to mówimy, że *dopasowuje się* on do tego wyrażenia (wyrażenie jest *wzorcem*, do którego staramy się *dopasować* podany napis). W praktyce powszechnie wykorzystuje się wyrażenia regularne w celu wyszukiwania napisów w plikach tekstowych. Również wydając DOS-owe polecenie `del *.*` w istocie piszemy wyrażenie regularne (w bardzo prostej notacji). Podobnie jak dla gramatyk bezkontekstowych, notacja matematyczna nie jest zbyt wygodna w zastosowaniach praktycznych. W Unix-ie mamy dostęp do plików nagłówkowych `regexp.h` i `regexpr.h` oraz pliku nagłówkowego `libgen.h` i modułu bibliotecznego `libgen.a` zawierających zbiór C-owych funkcji do operowania na wyrażeniach regularnych zapisanych w specjalnej notacji, tzw. *Internationalized Basic and Extended Regular Expressions*. Wiele programów dostępnych pod Unix-em zostało napisanych z użyciem tych bibliotek. Poniżej znajduje się opis fragmentu tej notacji. Mówi się na tę notację „wyrażenia regularne" (RE, od *regular expression*), choć nazwa jest nieco myląca: nie wszystkie języki regularne daje się nią opisać. Niektóre opisywalne nią języki nie są regularne, ani nawet bezkontekstowe!

W plikach tekstowych zwykle wyróżnia się *wiersze*, tj. ciągi znaków zakończone znakiem *nowego wiersza* (LF). Omówione niżej wyrażenia opisują ciągi znaków w ramach jednego wiersza.

Rozróżniamy *jednoznakowe* RE i dowolne RE. Jednoznakowe RE opisują zbiory słów jednoliterowych:

1. Do dowolnego znaku różnego od `.*[\^$` dopasowuje on sam.

2. Do napisu \c dopasowuje się znak *c*, gdzie *c* jest jednym spośród znaków .*[\^$.
3. Do kropki . dopasowuje się dowolny znak różny od znaku nowego wiersza.
4. Do niepustego ciągu znaków w nawiasach kwadratowych [] nie zaczynającego się znakiem ^ dopasowuje się jeden spośród wymienionych znaków. Jeśli po [występuje ^, to ten ciąg reprezentuje jeden znak *różny* od wymienionych i różny od znaku nowego wiersza. Napis *c*-*d* wewnątrz nawiasów [] reprezentuje wszystkie znaki o kodach od kodu znaku *c* do kodu znaku *d* włącznie, np. [a-zA-Z] reprezentuje dowolną małą lub wielką literę. Znak - traci swoje specjalne znaczenie, jeśli jest pierwszym lub ostatnim znakiem w ciągu ujętym w nawiasy []. Podobnie jeśli znak] występuje bezpośrednio po [, to nie zamyka ciągu, tylko reprezentuje sam siebie, np. [][] oznacza jeden ze znaków [i].

Dowolne RE tworzy się z jednoznakowych RE z pomocą następujących reguł:

1. Jednoznakowe RE jest RE.
2. Jednoznakowe RE, po którym następuje gwiazdka * oznacza ciąg dowolnej długości znaków opisanych przez to jednoznakowe RE, np. a* opisuje zbiór ciągów złożonych z litery a.
3. Jednoznakowe RE, po którym następuje \{*m*\} lub \{*m*,\} lub \{*m*,*n*\}, gdzie gdzie *m* i *n* są liczbami z przedziału 0 . . . 255, oznacza ciąg odpowiednio: dokładnie *m*, co najmniej *m* oraz co najmniej *m* oraz co najwyżej *n* wystąpień znaku opisywanego przez jednoznakowe RE (zatem {0,} jest równoważne napisaniu gwiazdki, zaś [A-Z]{3}[0-9]{4} opisuje stare polskie tablice rejestracyjne).
4. Konkatenacja RE opisuje słowa będące konkatenacjami słów opisywanych przez te RE, np. a*b* oznacza słowa złożone z pewnej liczby liter a, po których następuje pewna liczba liter b, zaś [tT]o[mM]asz opisuje zbiór czterech słów tomasz, Tomasz, toMasz i ToMasz.
5. Ujęcie RE w nawiasy \(i \) nie zmienia jego znaczenia.
6. Do napisu postaci *n*, gdzie *n* jest pojedynczą cyfrą różną od zera, dopasowuje się napis, który dopasował się do *n*-tego wyrażenia objętego nawiasami \(i \), np.

   ```
   ^\(.*\)\1$
   ```

   dopasuje się do każdego wiersza złożonego z dwóch takich samych słów (uwaga: ten język nie jest nawet bezkontekstowy!)
7. Znak ^ na początku RE oznacza, że do RE musi dopasować się początek wiersza, znak $ zaś, że koniec.

Dla przykładu [A-Za-z_][A-Za-z_0-9]* opisuje zbiór identyfikatorów w języku C, zaś literały całkowitoliczbowe w C są opisane wyrażeniami:

```
[1-9][0-9]*[uUlL]\{0,1\}                    (dziesiętne),
[0-7][0-7]*[uUlL]\{0,1\}                    (ósemkowe),
0[xX][0-9a-fA-F][0-9a-fA-F]*[uUlL]\{0,1\}   (szesnastkowe).
```

Z RE korzysta wiele programów Unix-owych: edytory tekstu (`ed`, `ex`, `vi`, `emacs`), programy wyszukujące (`grep`, `egrep`), programy przetwarzające teksty (`awk`, `sed`, `cut`, `paste`) i in. Używa się ich również do opisu leksemów języka w programach automatycznie generujących leksery (`lex`, `flex`, `ML-lex`).

Przykład 3.7. Załóżmy, że napisaliśmy tekst o pewnym wykładowcy, nazywając go w tekście wielokrotnie po imieniu. W pliku występują zatem słowa `ToMasz`, `ToMasza`, `ToMaszowi`, `ToMaszem`, `ToMaszu` itp. Z pewnych (nieistotnych tutaj) względów chcemy zmienić treść tekstu w taki sposób, by zastąpić `ToMasz`-a dwiema innymi osobami: `Jakub`-em i `Piotr`-em, ale z zachowaniem właściwej deklinacji. Wydanie w edytorze `vi` polecenia

```
%s/ToMasz\([a-z]*\)/Jakub\1 i Piotr\1/g
```

spowoduje zastąpienie w całym edytowanym pliku słów: `ToMasz` przez `Jakub i Piotr`, `ToMasza` przez `Jakuba i Piotra`, `ToMaszowi` przez `Jakubowi i Piotrowi`, itd.

Znak `%` na początku polecenia oznacza, że będzie ono dotyczyć wszystkich wierszy pliku, podobnie znak `g` (*global*) na końcu oznacza, że polecenie będzie dotyczyć wszystkich wystąpień wzorca w danym wierszu. Litera `s` po `%` oznacza komendę „zamień” (*substitute*), po której, ograniczone znakami `/` występują: wzorzec i tekst zamiany. Wzorzec `ToMasz\([a-z]*\)` oznacza słowo `ToMasz`, po którym nastepuje dowolny ciąg liter. Ponieważ ujęliśmy go w nawiasy `\(` i `\)`, będzie się można później do niego odwołać z pomocą napisu `\1`. W ten sposób zapamiętujemy (być może pustą) końcówkę. Po znaku `/` następuje tekst, który ma zostać wstawiony w miejsce napisu, który dopasował się do wzorca.

Dzięki RE nawet poważne zmiany tekstu mogą być przeprowadzone automatycznie i bardzo sprawnie. Jeśli tekst liczy kilkanaście stron, to wyszukiwanie słowa `ToMasz` (nawet z pomocą funkcji *Find* edytora) i ręczna zamiana na dwa inne imiona może zająć wiele czasu. Użycie opisanego wyżej polecenia pozwala uporać się z zadaniem w czasie poniżej 15 sekund (tyle potrzeba na wpisanie polecenia; komputer wykonuje je błyskawicznie).

Implementacja procedur wyszukiwania wzorca według powyższych wyrażeń jest podobna do opisanej w podrozdziale 3.7.3: wyrażenie regularne jest zamieniane na automat (zakodowany jako specjalny napis), który jest następnie symulowany (tak, jak to robiliśmy w podrozdziale 3.7.2). Procedura budowania automatu jest podobna do opisanej w podrozdziale 3.7.3, jednak nieco bardziej skomplikowana, wyszukujemy bowiem nie jedno konkretne słowo, tylko dowolne słowo pasujące do wzorca (wyrażenia regularnego).

3.8. Struktura leksykalna języków programowania

Opisując strukturę leksykalną języków zwykle dzieli się zbiór znaków na kilka (częściowo) rozłącznych klas. Niektóre znaki nie wchodzą w skład tokenów, jedynie pomagają w czytelnym sformatowaniu tekstu programu. Są to tak zwane *białe znaki*. Pewien zbiór znaków (zwykle zawierający cyfry, znaki `"`, `'` i inne) służy do budowy *literałów* (napisów oznaczających stałe), litery i cyfry tworzą *identyfikatory* (ciągi znaków oznaczające zmienne) i *słowa kluczowe* (specjalne ciągi znaków oznaczające pewne pojęcia w języku). Pozostałe znaki, to tzw. *znaki przestankowe* (pomocnicze).

3.8.1. Sposób formatowania tekstu programu

Języki programowania dzielą się na *języki o zapisie pozycyjnym* (*sztywnym*) i *języki o zapisie liniowym* (*swobodnym*). W tych pierwszych, przeważających w przeszłości, zapis programu rządzi się ścisłymi regułami.

Przykład 3.8. W FORTRAN-ie przyjęto, że pięć pierwszych znaków każdego wiersza może zawierać etykietę instrukcji (ciąg cyfr), jeśli zaś pierwszym znakiem w wierszu jest `*` lub `C` oznacza to, że w danym wierszu występuje komentarz (tzn. ten wiersz ma być zignorowany przez kompilator), kolumny od 7 do 72 zawierają instrukcję (co najwyżej jedną w wierszu), znaki w dalszych kolumnach są ignorowane przez kompilator. Szósty znak w wierszu, w którym zaczyna się instrukcja musi być spacją lub zerem, wiersze zawierające ciąg dalszy instrukcji muszą zawierać na szóstej pozycji znak różny od spacji i `0` i muszą mieć spacje na pozycjach od pierwszej do piątej. Instrukcja może znajdować się w co najwyżej 20 kolejnych wierszach.

Takie ustalenia znakomicie ułatwiają analizę leksykalną tekstu programu i równie znakomicie utrudniają pracę programiście. Począwszy od Algolu 60 w większości współczesnych języków programowania, tzw. *językach o strukturze liniowej*, przyjęto więc przeciwne rozwiązanie: program jest ciągiem tokenów swobodnie rozdzielanych *białymi znakami*. Białe znaki nie mogą występować we wnętrzu tokenu (wyjątek stanowią literały łańcuchowe, tam jednak białe znaki zatracają swoje znaczenie). Niekiedy są wymagane dla oddzielenia dwóch tokenów od siebie. Występowanie białych znaków w tekście programu nie ma wpływu na jego znaczenie. Pojęcie wiersza programu albo nie jest wprowadzane wcale, albo ma drugorzędne znaczenie (np. przy definiowaniu literałów łańcuchowych i niektórych komentarzy). Niektóre współczesne języki o strukturze liniowej zawierają pewne elementy struktury pozycyjnej (np. Occam, Haskell i Concurrent Clena).

Przykład 3.9. W Haskellu deklaracja funkcji może mieć postać:

nazwa-funkcji zmienne `=` *wyrażenie* `where` *deklaracja-pomocnicza*

Jeżeli chcielibyśmy napisać dwie *deklaracje-pomocnicze*, musimy objąć je nawiasami klamrowymi:

```
f x y = u + v + 1 where
          {
            u = x*x;
            v = y*y
          }
```

Jeśli jednak opuścimy nawiasy i średnik:

```
f x y = u + v + 1 where
          u = x*x
          v = y*y
```

wówczas są one odtwarzane według następującej reguły: częścią deklaracji funkcji są wszystkie deklaracje, *które się zaczynają w kolumnie o nie mniejszym numerze niż kolumna, w której zaczyna się pierwsza deklaracja po słowie* `where`. Dla przykładu dwa ostatnie programy są równoważne, podczas gdy w programie

```
f x y = u + v + 1 where
          u = x*x
   v = y*y
```

w zasięgu deklaracji funkcji znajduje się jedynie deklaracja `u = x*x`, podczas gdy deklaracja `v = y*y` jest umieszczona na globalnym poziomie programu. Programiści, których irytuje zapis pozycyjny powinni zawsze otaczać nawiasami nawet pojedynczą deklarację po słowie `where`, wówczas program jest analizowany tak, jak w języku o strukturze liniowej.

3.8.2. Białe znaki i komentarze

Białe znaki to najczęściej *spacje*, *znaki tabulacji* i *znaki nowego wiersza*. Komentarze są traktowane jak białe znaki i są zwykle otaczane specjalnymi ciągami znaków, np. `(*` i `*)` oraz `{` i `}` w Pascalu, (te pierwsze również w SML-u), `/*` i `*/` w C, `{-` i `-}` w Haskellu i mogą obejmować wiele wierszy. Komentarze mogą być *zagnieżdżane* lub nie. W pierwszym przypadku, np. w SML-u, procedura pomijania komentarzy w kompilatorze nieco się komplikuje, jednak programista może łatwo „zasłaniać" pewne fragmenty tekstu programu (być może już zawierające komentarze), jeśli nie chce ich przedstawiać do kompilacji. W drugim przypadku, np. w C, kosztem wygody programisty upraszcza się zarówno definicja komentarza: komentarz to dowolny ciąg znaków zaczynający się znakami `/*`, nie zawierający wewnątrz znaków `*/` i zakończony znakami `*/` (wewnątrz natomiast może zawierać znaki `/*`, nie zaczynają one jednak nowego komentarza), zatem komentarzem jest np.

```
/* to jest /* komentarz */
```

Wiele języków dopuszcza również krótkie, jednowierszowe komentarze, zaczynające się np. znakami `%` w TEX-u i Prologu, `//` w C++ czy `--` w Adzie i Haskellu. W wierszu, w którym występują takie znaki, jako komentarz uznaje się tekst zaczynający się tymi znakami i zakończony znakiem nowego wiersza. Przydają się one do umieszczania krótkich uwag w treści programu.

3.8.3. Identyfikatory i słowa kluczowe

W każdym języku programowania ustala się pewien zbiór znaków, z których mogą być zbudowane identyfikatory, zwykle wielkie i małe litery `A` ÷ `Z` i `a` ÷ `z`, cyfry `0` ÷ `9` i pewne dodatkowe znaki, z których najpopularniejszym jest znak podkreślenia „_". Ponieważ zbiór znaków tworzących identyfikatory zawiera cyfry używane również do budowy literałów, zakłada się, że identyfikatory muszą zaczynać się literą (a literały nie mogą zaczynać się literą). *Słowa kluczowe* to pewne wyróżnione tokeny takiej samej postaci jak identyfikatory. W niektórych językach wolno używać identyfikatorów będących słowami kluczowymi (wówczas

nie wyróżnia się tokenów opisujących słowa kluczowe, a o rodzaju danego napisu decyduje kontekst jego użycia opisany gramatyką języka). Dla przykładu w PL/I wolno napisać

```
IF IF = THEN THEN THEN = ELSE ELSE ELSE = THEN;
```

(instrukcja rodem z koszmarnego snu programisty poprawiającego program napisany przez kogoś innego). Z widocznych powyżej powodów w wielu językach programowania słowa kluczowe są *zastrzeżone* i wykluczone ze zbioru identyfikatorów. Wówczas zadaniem leksera jest sprawdzić, czy dany token występuje wśród słów kluczowych, czy nie i odpowiednio go zaklasyfikować.

W SML-u zmienne mają taką samą postać jak konstruktory. Ponieważ zbiór konstruktorów może się zmieniać, przyjęto, że wszystko, co nie jest w danym kontekście konstruktorem, jest uważane za zmienną. To może prowadzić do podobnych problemów, jak w przypadku mylenia identyfikatorów i słów kluczowych. Dlatego np. w Haskellu i Concurrent Cleanie przyjęto, że zbiór identyfikatorów rozpada się na rozłączne klasy słów kluczowych, konstruktorów i zmiennych (konstruktory pisze się koniecznie wielką literą, zmienne — małą). Ogranicza to nieznacznie swobodę programisty, ale znakomicie zwiększa czytelność programu i jego odporność na błędy. Umożliwia też lekserowi łatwe sklasyfikowanie wczytanego tokenu do jednej z trzech powyższych kategorii.

W niektórych językach (np. w Pascalu) przyjęto, że wielkie i małe litery w identyfikatorach są utożsamiane (ten pomysł pochodzi z czasów, gdy niektóre komputery oferowały jedynie możliwość pisania wielkimi literami, na długo przed wprowadzeniem standardu ASCII). W nowoczesnych językach odchodzi się od tej tradycji. Również w C wielkie i małe litery są rozróżniane.

Ze względów oszczędnościowych w dawnych językach ograniczano dopuszczalną długość identyfikatorów. Np. w FORTRAN-ie przyjęto, że identyfikator może się składać z co najwyżej sześciu znaków, zatem identyfikator `LICZBA_ITERACJI` był odrzucany jako niepoprawny. W innych językach dopuszczano identyfikatory dowolnej długości, ale zakładano, że tylko pewna liczba (np. 16) początkowych znaków jest rozpoznawana przez kompilator. Wówczas np. identyfikatory

```
saldo_rachunku_bankowego_najbogatszego_klienta
saldo_rachunku_bankowego_autora_tego_programu
```

były akceptowane przez kompilator, ale traktowane jako

```
saldo_rachunku_b
```

i utożsamiane. Było to źródłem wielu trudnych do wykrycia błędów. Moc obliczeniowa współczesnych komputerów jest wystarczająca, by kompilator mógł sobie poradzić z identyfikatorami praktycznie dowolnej długości (np. kompilator SML/NJ akceptuje identyfikatory długości nawet 2000 znaków — dla dłuższych nie sprawdzałem).

W różnych językach są popularne różne konwencje wyboru identyfikatorów.

Przykład 3.10. W Adzie preferuje się zwykle długie nazwy (nawet powyżej 20 znaków) i pisze się je wielkimi literami (aby je wizualnie odróżnić od słów kluczowych, zapisywanych

małymi literami). Ada (nawet w porównaniu z Pascalem) ma również dosyć długie słowa kluczowe i preferuje się w niej alfanumeryczne słowa kluczowe zamiast symboli specjalnych (np. `return` zamiast Pascalowego `:`, `is` zamiast `=`, itp.), co widać w poniższej deklaracji funkcji:

```
function SPRAWDZ_DZIALANIE_URZADZENIA (STEROWNIK : URZADZENIE)
      return BOOLEAN is
   begin
      return STEROWNIK.STAN = ON_LINE;
   end SPRAWDZ_DZIALANIE_URZADZENIA;
```

W języku C, w którym zapis jest bardziej zwięzły, preferuje się krótsze identyfikatory (pisane zwykle małymi literami; wielkimi pisze się stałe) i używa się symboli zamiast słów kluczowych:

```
int sprUrzadz (urzadz sterownik) {
   return sterownik.stan == ON_LINE;
}
```

Przesadnie zwięzły zapis może utrudnić zrozumienie programu.

Przykład 3.11. „Rekordzistą” w zwięzłości był język APL/360, którego zestaw znaków był tak wielki, że wymagał specjalnych klawiatur i drukarek. Program drukujący trójkąt Pascala można było napisać w sześciu krótkich linijkach ([140], str. 169):

```
∇ PASCAL
P←,1
SPACJE←(0⌈SRODEKWIERSZA-⌊0.5×+/3+⌊10⊛P)⍴' '
SPACJE;P
→2×≥⍴P←(0,P)+P,0
∇
```

jednak czas stracony na jego zrozumienie jest pewnie dłuższy od zaoszczędzonego przy wpisywaniu jego tekstu do komputera.

Podobnie „zwięzłe” programy można pisać dla Unix-owego kalkulatora programowalnego `bc`, który używa notacji postfiksowej, a wszystkie jego instrukcje są jednoliterowe. Program drukujący wartości funkcji silnia dla liczb $1, \ldots, 10$ można w nim zapisać następująco (Sun OS 5.7 Manual for `bc`):

```
[la1+dsa*pla10>y]sy
0sa1
lyx
```

Z powyższych przykładów wynika, że pewna nadmiarowość znaków w programie jest potrzebna, gdyż zwiększa czytelność tekstu, a tym samym zmniejsza ryzyko błędów. Znaczna część społeczności programistów uważa, że przykładem właściwej równowagi pomiędzy

zwięzłością i czytelnością programu jest język C. Składnia wielu innych języków jest więc na nim wzorowana.

Właściwy wybór identyfikatorów jest bardzo ważny z punktu widzenia czytelności programu. Identyfikator powinien jednoznacznie sugerować znaczenie wartości, którą nazywa. W programie bankowym instrukcja

```
Z = X+Y;
```

jest dużo mniej czytelna, niż

```
saldoPoKapital = saldoPoprz + odsetki;
```

Z wymyślaniem „znaczących" nazw zmiennych nie należy jednak przesadzać. Niekiedy dla argumentu funkcji trudno wymyślić sensowniejszą nazwę niż po prostu `x`, np. w definicji funkcji identyczności w SML-u:

```
fun id x = x
```

Najczęściej używaną strukturą danych w SML-u są listy. W języku angielskim liczbę mnogą rzeczownika otrzymuje się najczęściej przez dopisanie na jego końcu litery `s`. Niektórzy programiści SML-owi przyjęli konwencję, że anonimowe wartości oznacza się literami `x`, `y`, `z`, ..., listy takich wartości — identyfikatorami `xs`, `ys`, `zs`, ... (`xs` znaczy „iksy", „iks" w liczbie mnogiej, dużo „iksów", tj. ich lista), listy list — identyfikatorami `xss`, `yss`, `zss`, ...

Dawniej, gdy identyfikatory musiały być bardzo krótkie, obmyślano zestawy reguł, według których je tworzono, np. że należy napisać pełną nazwę danego obiektu (być może wielowyrazową), pominąć spójniki, przedimki itp., uwzględnić co najwyżej trzy pierwsze wyrazy, usunąć ich końcówki i tyle samogłosek i mniej ważnych spółgłosek poczynając od końca, by otrzymany identyfikator był dostatecznie krótki ([166], str. 19–24). Obecnie, gdy identyfikatory mogą mieć nawet kilkanaście znaków, tak ścisłe reguły nie są potrzebne. Warto jednak wypracować swój własny styl tworzenia identyfikatorów i go przestrzegać. Poszczególne wyrazy warto oddzielać znakiem podkreślenia lub pisać wielką literą, by nie zlewały się w jedną całość. Zabawnym przykładem na użyteczność znaku _ jest identyfikator `poradnia` występujący w pewnym programie obsługującym kalendarz.[7]

3.8.4. Literały

Do zapisu *stałych* w języku używa się *literałów*. Zwykle są to literały całkowitoliczbowe, zmiennopozycyjne (zmiennoprzecinkowe), znakowe i łańcuchowe (napisowe). W większości współczesnych języków używa się konwencji zapisu literałów mniej lub bardziej zbliżonych do konwencji przyjętych w języku C.

[7] Dużo czytelniej było by napisać `PoraDnia` lub `pora_dnia`.

3.8.5. Zasada zachłanności

Patrząc z poziomu leksykalnego program jest ciągiem tokenów. Zadaniem leksera jest podzielić ciąg znaków tworzących program na tokeny. Nie zawsze jest to jednoznaczne. Dla przykładu C-owe wyrażenie `x+++y` może oznaczać `x++ +y`, gdzie `++` jest operatorem postinkrementacji a `+` — dodawania, lub `x+ ++y`, gdzie `++` jest operatorem preinkrementacji. Aby umożliwić jednoznaczny rozbiór leksykalny ciągu znaków `x+++y` przyjęto w definicji języka C, że *token rozciąga się tak daleko na prawo, jak tylko można go sensownie przeczytać*. Jest to tzw. *zasada zachłanności*. Dlatego `+++` zostanie przeczytane jako para tokenów `++` po którym następuje `+`. Ta zasada jest ślepo stosowana przez lekser nawet wówczas, gdy *sensowny* program można otrzymać tylko na jeden sposób. Np. wyrażenie `---x` zostanie przeczytane jako `-- -x` i odrzucone przez kompilator jako niepoprawne (operator predekrementacji może być użyty jedynie w odniesieniu do zmiennych), podczas gdy potencjalnie mogłoby być przeczytane jako `- --x`, gdyby zasada zachłanności nie obowiązywała.

Zasada zachłanności wymusza stosowanie białych znaków do odzielenia tokenów, które mogłyby się w przeciwnym razie „zlać” w jedną całość. Np. w Pascalu wolno napisać wyrażenie `12mod 3`, jednak `12mod3` nie jest poprawnym wyrażeniem, ponieważ składa się z literału całkowitoliczbowego i identyfikatora `mod3`.

3.8.6. Struktura leksykalna języka w praktyce

Napisanie dobrego programu wymaga nie tylko wiedzy teoretycznej na temat programowania, lecz również dobrego opanowania programistycznego rzemiosła. Wiele cennych uwag o tym, jak wykorzystać strukturę leksykalną języka do napisania dobrego programu zawiera rozdział 1 książki Tassela [166], str. 11–47.

3.8.7. Struktura leksykalna języka Standard ML

Identyfikatory w SML-u, podobnie jak w innych językach programowania, służą do nazywania różnych obiektów: 1) wartości (zmiennych, konstruktorów, wyjątków),[8] 2) typów, 3) zmiennych typowych, 4) pól rekordów, 5) sygnatur, 6) struktur i 7) funktorów.[9] Identyfikatory zmiennych typowych mają specjalną postać, pozostałe klasy identyfikatorów występują w rozłącznych kontekstach (tj. miejsce wystąpienia identyfikatora określa, do której klasy należy), dlatego można używać tych samych identyfikatorów na oznaczenie obiektów z różnych klas, np. `real` jest nazwą predefiniowanego typu danych, a także nazwą funkcji standardowej. W języku C jest to zabronione.

W identyfikatorach wielkie i małe litery są rozróżnialne (np. następujące identyfikatory są parami różne: `tomasz`, `Tomasz` i `ToMasz`).

Identyfikatory mogą być dwóch rodzajów: *alfanumeryczne* i *symboliczne*. Te pierwsze znane są z Pascala i C. W SML-u są to niepuste ciągi wielkich i małych liter, cyfr oraz

[8] O wyjątkach, typach i zmiennych typowych oraz sygnaturach, strukturach i funktorach mówimy w dalszych rozdziałach skryptu.

[9] . . . i sygnatur funktorów w SML/NJ.

znaków apostrofu ’ i podkreślenia _, zaczynające się literą. W porównaniu z Pascalem i C, nowością jest apostrof („prim”). Dzięki temu można pisać tak jak w matematyce `x`, `x’`, `x’’` itp. Inaczej niż w C, identyfikator nie może zaczynać się znakiem podkreślenia.

Identyfikatory zmiennych typowych mają szczególną postać: zaczynają się jednym lub dwoma znakami apostrofu ’, po których następuje zwykły identyfikator alfanumeryczny. Wypisując odpowiedzi system tworzy własne nazwy zmiennych z kolejnych liter alfabetu, poczynając od `’a`, `’b` itd. Programista może oczywiście nadawać zmiennym dowolne nazwy, np.:

```
fun identycznosc (x : ’dowolny_typ) = x
```

Identyfikatory użyte do nazwania wszelkich innych obiektów poza zmiennymi typowymi muszą zaczynać się literą.

Identyfikatory symboliczne to dowolne niepuste ciągów następujących znaków:

```
! % & $ # + - * / : < = > ? @ \ ~ ‘ ^ |
```

Przykładami identyfikatorów symbolicznych są: `<=`, `**`, `*`, `/\` itd. Z początku trudno się do nich przyzwyczaić, ponieważ są rzadko spotykane w innych językach programowania. W połączeniu jednak z możliwością zmiany łączliwości operatorów (por. podrozdział 3.9.1) są bardzo użyteczne — można definiować operatory infiksowe o składni takiej samej jak ich standardowe odpowiedniki (np. można zdefiniować typ danych `complex` reprezentujący liczby zespolone i zdefiniować dla niego operator mnożenia, np. `**`, tak by można było pisać np. `x**y`).

W SML-u przyjęto zasadę zachłanności, tzn. założono, że pojedyncza jednostka leksykalna rozciąga się maksymalnie na prawo. O ile jest oczywiste, że dwa alfanumeryczne słowa kluczowe należy z tego powodu rozdzielić spacją (np. `letval` to identyfikator, zaś `let val` — para słów kluczowych), o tyle czasem może nie być jasne, że należy również rozdzielać słowa kluczowe symboliczne (ponieważ inaczej zleją się w jeden identyfikator symboliczny), np. `%=>` to pojedynczy identyfikator symboliczny, podczas gdy `% =>` — identyfikator symboliczny i słowo kluczowe. Ponieważ zbiory znaków z których buduje się identyfikatory symboliczne i alfanumeryczne są rozłączne, można je zestawiać obok siebie bez spacji, np. `x=>` to identyfikator `x` po którym następuje słowo kluczowe `=>`.

Słowa kluczowe SML-a są przedstawione w tablicy 3.7 na następnej stronie. Słowa kluczowe w SML-u są zastrzeżone i nie wolno ich używać jako identyfikatorów.[10] Zwróćmy uwagę na grupę słów kluczowych symbolicznych, które w szczególności nie mogą być używane jako identyfikatory symboliczne.

Pozostałe znaki używane w programach SML-owych to `{ } ( ) [ ] . , ; "`. Są to znaki przestankowe i nie mogą być częścią identyfikatorów. Nie trzeba ich oddzielać od identyfikatorów i słów kluczowych białymi znakami, choć często nieco spacji poprawia czytelność programu. Wewnątrz łańcuchów i komentarzy mogą pojawiać się dowolne znaki ASCII.

[10] W SML/NJ zastrzeżono dodatkowo `funsig`.

```
abstype and andalso as case datatype do else end eqtype
exception fn fun functor handle if in include infix
infixr let local nonfix of op open orelse raise rec
sharing sig signature struct structure then type val
where while with withtype : :> _ | = => -> #
```

Tablica 3.7. Słowa kluczowe języka Standard ML

3.9. Wyrażenia w językach programowania

3.9.1. Operatory infiksowe w SML-u

W SML-u operatory infiksowe mogą łączyć w lewo, (tj. $x \oplus y \oplus z = (x \oplus y) \oplus z$), lub w prawo (tj. $x \oplus y \oplus z = x \oplus (y \oplus z)$), i mieć priorytet od 0 do 9 (silniej wiążą operatory o wyższym priorytecie). Użytkownik może zmieniać łączliwość operatorów dyrektywami

`infix` [*n*] *identyfikatory* `infixr` [*n*] *identyfikatory* `nonfix` *identyfikatory*

gdzie *n* jest pojedynczą cyfrą dziesiętną i oznacza priorytet (jeśli nie występuje, przyjmuje się, że priorytet wynosi 0), a *identyfikatory*, to lista identyfikatorów których łączliwość zmieniamy (wypisanych jeden po drugim bez przecinków itp., jedynie rozdzielonych spacjami). Dyrektywa `infix` oznacza, że wymienione identyfikatory stają się operatorami infiksowymi łączącymi w lewo, `infixr` — w prawo, a `nonfix` — że (dotychczas być może operatory infiksowe) stają się identyfikatorami prefiksowymi. Na przykład:

```
fun plus (x:int,y) = x + y;
- plus (2,5);
val it = 7 : int
```

`plus`, jak każdy „nowy” identyfikator jest z początku prefiksowy. Nie możemy go użyć infiksowo:

```
- 2 plus 5;
Error: operator is not a function
  operator: int
  in expression:
    2 plus
```

Dyrektywą `infix` zmieniamy jego łączliwość i możemy go wówczas użyć jako operatora infiksowego:

```
- infix 5 plus;
infix 5 plus
- 2 plus 5;
val it = 7 : int
```

Niestety w obszarze obowiązywania dyrektywy `infix`, identyfikator `plus` nie może być używany jako operator prefiksowy. Możemy jednak powtórnie zmienić jego łączliwość dyrektywą `nonfix`:

```
- plus (2,5);
Error: nonfix identifier required
- nonfix plus;
nonfix plus
- plus (2,5);
val it = 7 : int
```

Aby jednorazowo wskazać, że używamy operatora infiksowego w postaci prefiksowej, nie trzeba używać dyrektywy `nonfix`: wystarczy w miejscu użycia identyfikatora napisać przed nim słowo kluczowe `op`:

```
- infix plus;
infix plus
- op plus (2,5);
val it = 7 : int
```

Jest to szczególnie wygodne, gdy chcemy skorzystać ze standardowego operatora infiksowego jako ze zwykłej funkcji nie zmieniając globalnie jego łączliwości, np. `op+ (2,5)` znaczy to samo co 2+5.

Zasięg dyrektyw `infix`, `infixr` i `nonfix` jest ograniczony wyrażeniem `let` i deklaracją `local`. Jeśli chcemy więc zmienić łączliwość jakiegoś operatora tylko w pewnym kontekście, wygodnie jest użyć powyższych konstrukcji do ograniczenia zasięgu tej zmiany, np.

```
- let
=   nonfix +
= in
=   + (2,5)
= end;
val it = 7 : int
```

Za słowem `end` kończącym wyrażenie `let`, operator `+` dalej posiada swoją standardową łączliwość.

Łączliwość identyfikatora można zmienić także wówczas, gdy dotychczas nie pojawiał się w programie. Wówczas jednak musimy konsekwentnie używać go w nowej składni również w definicjach, np.

```
- infixr 9 **;
infixr 9 **
- fun x ** y =
=   if y <= 0
=     then 1
=     else x * x**(y-1);
val ** = fn : int * int -> int
```

Zwróćmy uwagę, że po słowie `fun` identyfikator `**` (tu wykorzystany do nazwania funkcji potęgowania) występuje również w postaci infiksowej. Alternatywnie można by napisać:

```
fun op** (x,y) = ...
```

lecz nie jest to zbyt zgrabne.

Zmiana łączliwości identyfikatora dotyczy jedynie identyfikatorów nazywających wartości, np. po wykonaniu dyrektywy `nonfix *` identyfikator `*` oznaczający krotkę w wyrażeniach typowych dalej jest infiksowy.

Użycie dwóch operatorów o tym samym priorytecie i różnych kierunkach łączności jest zabronione przez standard języka, zaś kompilator wypisuje ostrzeżenie i traktuje oba operatory tak, jakby łączyły w lewo:

```
- infix 5 +;
infix 5 +
- infixr 5 -;
infixr 5 -
- 4+3-2;
Warning: mixed left- and right-associative operators of same
         precedence
val it = 5 : int
```

Na koniec jeszcze jeden przykład ilustrujący użyteczność operatorów infiksowych — leksykograficzny porządek na parach liczb naturalnych:

```
- infix <<
= fun (x1:int,y1:int) << (x2,y2) =
=     x1 < x2 orelse (x1=x2 andalso y1<y2);
infix <<
val << = fn : (int * int) * (int * int) -> bool
- (1,2) << (3,4);
val it = true : bool
```

Dyrektywy `infix` powodują, że składnia języka ulega zmianie w trakcie kompilowania programu. Jest to wyzwanie dla parsera, który zwykle ma składnię „zaszytą” w swojej strukturze. W kompilatorze SML/NJ rozwiązano ten problem następująco: parser traktuje wszystkie wyrażenia jako ciągi tokenów (nie dokonuje ich analizy), tylko przekazuje je do osobnego analizatora ze stosem, podobnego do opisanego w podrozdziale 3.6.5. Dyrektywy `infix` modyfikują tablice priorytetów tego analizatora.

Dzięki potraktowaniu operatorów jako zwykłych identyfikatorów symbolicznych lub alfanumerycznych i przyjęciu opisanych wyżej mechanizmów ustalania priorytetów i kierunków łączności, definicja składni wyrażeń w języku SML jest bardzo zwięzła (mniej niż dwie strony opisu). Nie trzeba w szczególności w definicji języka opisywać wszystkich operatorów predeklarowanych (są one opisane w definicji środowiska standardowego). Dla porównania składnia wyrażeń w definicji języka C++ zajmuje 19 stron i w osobnych paragrafach omawia operatory addytywne `+` i `-`, operatory multiplikatywne `*`, `/` i `%`, itd.

3.9.2. Wyrażenia w innych językach programowania

Pascal i C. W językach takich jak C lub Pascal zbiór operatorów jest ustalony i są one traktowane jako symbole specjalne. Programista nie może definiować własnych operatorów. Operatory podzielone są zwykle na kilka grup o różnych priorytetach (w Pascalu 4, w C zaś aż 16) i różnych kierunkach łączności. W Pascalu priorytety są ustalone niezbyt wygodnie (w szczególności jest ich zbyt mało), gdyż np. w wyrażeniu `(x<y) and (x<z)` nawiasów opuścić nie wolno. Aż 16 poziomów wyrażeń w C komplikuje natomiast opis składni języka i znacznie utrudnia korzystanie z reguł opuszczania nawiasów, bo nie sposób ich wszystkich spamiętać. Dla porównania w SML-u i Haskellu mamy 10 poziomów operatorów, w Adzie zaś 6. Wydaje się, że są to liczby optymalne (w Adzie priorytetów jest mniej, ponieważ operatory infiksowe nie są tak powszechnie używane, jak w SML-u).

C++ i Ada. W językach C++ i Ada operatory (zarówno infiksowe, jak i pre- oraz postfiksowe) również są symbolami specjalnymi i ich zestaw jest ustalony. W C++ są to:

```
+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<=
== != <= >= && || ++ -- ->* , [] () new delete
```

w Adzie zaś:

```
& * ** + - / /= < <= = > >= abs and mod not or rem xor
```

W obu tych językach można jednak *przeciążać* operatory, tj. używać istniejące operatory do nazywania funkcji działających na argumentach różnych typów. Nie można natomiast zmienić ich arności, priorytetów, ani kierunków łączności (zatem np. w C++ operator `!` musi być unarny prefiksowy). Nie można również definiować nowych operatorów. To jest zupełnie sprzeczne z filozofią przyjętą w SML-u, gdzie zachęca się programistę do tworzenia nowych operatorów, np. `**` lub `/\`, natomiast odradza się wykorzystywania do innych celów operatorów standardowych. W SML-u nie ma natomiast możliwości przeciążania operatorów przez programistę.

Haskell. W Haskellu przyjęto podobne rozwiązania jak w SML-u, w szczególności zdefiniowano również dwie klasy identyfikatorów: alfanumeryczne i symboliczne, jednak jest stały podział na identyfikatory prefiksowe i infiksowe: identyfikatory alfanumeryczne są prefiksowe, zaś identyfikatory symboliczne — infiksowe. Jeżeli programista pragnie użyć identyfikatora alfanumerycznego jako operatora infiksowego, musi go ująć w odwrócone apostrofy, np. ‘`mod`‘ jest Haskellowym operatorem dzielenia całkowitego. Priorytety i kierunki łączności operatorów infiksowych można zmieniać podobnymi dyrektywami jak w SML-u:

(`infix` | `infixl` | `infixr`) [*cyfra*] *identyfikator* [{`,`*identyfikator*}]

(`infix` oznacza, że operator nie jest łączny, `infixl`, że wiąże w lewo, `infixr` zaś, że w prawo). Nie ma dyrektywy `nonfix`, ponieważ podział na operatory infiksowe i prefiksowe jest stały i programista nie może go trwale zmienić. Identyfikatory oddziela się przecinkami. Znaczenie priorytetu jest takie, jak w SML-u. Nie ma konstrukcji `op`, która w SML-u

pozwala na chwilowe użycie operatora infiksowego jako zwykłego identyfikatora prefiksowego, jest jednak w zamian dużo ogólniejszy mechanizm tzw. *sections*. Rozważmy wyrażenie $x \oplus e_2$, w którym głównym operatorem jest $\oplus$, x jest zmienną, a e_2 pewnym wyrażeniem. Napis $(\oplus e_2)$ oznacza w Haskellu funkcję, która argumentowi x przyporządkowuje wartość $x \oplus e_2$. Podobnie $(e_1 \oplus)$ oznacza funkcję, która argumentowi x przyporządkowuje wartość $e_1 \oplus x$. W końcu $(\oplus)$ oznacza funkcję, która argumentowi x przyporządkowuje wartość $(x \oplus)$. Zatem ujęcie operatora w nawiasy ma takie samo znaczenie, jak użycie słowa kluczowego `op` w SML-u — sprawia, że operator jest w wyrażeniu traktowany jak identyfikator prefiksowy. Dla przykładu `(+) 3 5` jest poprawnym wyrażeniem Haskellowym (i ma wartość 8), zaś `(+1)` jest funkcją następnika. Jedynym wyjątkiem jest operator `-`, który może być zarówno unarny, jak i binarny. Wyrażenie `(-1)` nie jest funkcją poprzednika, tylko oznacza liczbę -1.

Prolog. W Prologu do zmiany łączliwości operatorów służy predykat

`op`(*priorytet*, *typ*, *nazwa*)

gdzie *priorytet* jest nieujemną liczbą całkowitą nie większą niż 1200 (tak mówi standard, górna granica bywa w niektórych kompilatorach inna), a *nazwa* jest nazwą operatora, którego łączliwość zmieniamy (lub listą nazw operatorów). Priorytet ma odwrotne znaczenie niż np. w SML-u: najsilniej wiąże operator o priorytecie 0, najsłabiej operator o priorytecie 1200. Pojęcie priorytetu rozszerza się na wyrażenia: wyrażenie atomowe ma priorytet 0, podobnie wyrażenie ujęte w nawiasy. Priorytetem wyrażenia będącego aplikacją pewnego operatora do argumentów jest priorytet tego operatora. *Typ* opisuje łączliwość operatora i jest jednym z napisów: `xf`, `yf`, `xfx`, `xfy`, `yfx`, `yfy`, `fy` i `fx` i ma następujące znaczenie: `f` oznacza operator, `x` i `y` jego argumenty. Zatem napis `fy` i `fx` oznaczają operator prefiksowy, `xf` i `yf` operator postfiksowy, zaś `xfx`, `xfy`, `yfx` i `yfy` operator infiksowy. Litera `x` oznacza wyrażenie, którego priorytet ma być ostro mniejszy od priorytetu operatora `f`, zaś `y` oznacza wyrażenie, którego priorytet jest mniejszy bądź równy priorytetowi operatora `f`. Zatem `xfy` oznacza operator łączący w prawo, zaś `yfx` operator łączący w lewo, `xfx` oznacza, że dwa takie operatory nie mogą wystąpić w jednym wyrażeniu (np. operator `<=` w Pascalu mógłby mieć taki typ, ponieważ wyrażenie `1 <= 2 <= 3` jest nielegalne), zaś `yfy` oznacza, że operator jest łączny. Podobnie typ `yf` zezwala na użycie więcej niż jednego operatora tego typu w wyrażeniu (np. można by napisać `- - 1`, jeśli `-` miałby typ `yf`), zaś typ `xf` zabrania takiego użycia.

Lisp. W Lispie konsekwentnie użyto zapisu prefiksowego i nie ma w nim wcale operatorów infiksowych. W większości języków można swobodnie używać nawiasów do grupowania wyrażeń (bardzo liberalny jest pod tym względem SML, który pozwala napisać zarówno `Math.sin (3.0)`, jak i `(Math.sin) 3.0`, i znaczy to to samo, co `Math.sin 3.0`). Lisp jest wyjątkowym językiem z tego powodu, że w nawiasy okrągłe mają nim specjalne znaczenie: służą do tworzenia list. Dlatego napisy `5` i `(5)` znaczą w Lispie co innego (to drugie jest *listą*, której elementem jest `5`). Wyrażenie w Lispie zapisujemy jako listę, której pierwszym

elementem jest operator, po którym następują jego argumenty. Całość musi być ujęta w jedną parę nawiasów. Przykładem wyrażenia lispowego jest `(RAZY (PLUS 5 7) 8)`. Notacja lispopodobna jest dosyć powszechnie używana do dziś (wykorzystują ją m.in. programowalne edytory tekstu, np. `emacs`).

FORTH i Postscript. W obu językach konsekwentnie używa się notacji postfiksowej, również do zapisu instrukcji. Nawiasy okrągłe mają specjalne znaczenie (w Postscripcie np. oznaczają tekst do wypisania) i nie wolno ich używać do grupowania wyrażeń. Postscript jest językiem wewnętrznym większości procesorów sterujących drukarkami i służy do opisu grafiki wektorowej.

Smalltalk i Algol 60. Ciekawe rozwiązanie składniowe przyjęto w obiektowym języku Smalltalk, mianowicie metody obiektu (tu zwane wiadomościami, *messages*) z parametrami są zapisywane w notacji miksfiksowej: nazwa metody składa się z kilku identyfikatorów (tu zwanych słowami kluczowymi, *keywords*[11]) zakończonych dwukropkiem, np.

```
prostokat wysokosc: 12 szerokosc: 15
```

oznacza wysłanie do obiektu `prostokat` wiadomości (wywołanie metody)

```
wysokosc:szerokosc:
```

z parametrami `12` i `15`. W „normalnym” języku programowania napisalibyśmy:

```
prostokat.WysokoscSzerokosc(12,15)
```

Taka notacja okazuje się całkiem czytelna. Występowała również w języku Algol 60, w którym składnia nagłówka procedury była następująca:

⟨nagłówek procedury⟩ ::= ⟨nazwa procedury⟩ ⟨zbiór parametrów⟩
⟨zbiór parametrów⟩ ::= ⟨puste⟩ | (⟨wykaz parametrów⟩)
⟨wykaz parametrów⟩ ::= ⟨parametr⟩ |
⟨wykaz parametrów⟩ ⟨ogranicznik⟩ ⟨parametr⟩
⟨ogranicznik⟩ ::= , |) ⟨łańcuch liter⟩ : (

zatem zamiast przecinka do oddzielania parametrów procedury można było użyć dłuższego napisu, np. postaci:

TransponujMacierz (*A*) *rozmiaru:* (*n*) *wynik wstaw do:* (*B*);

zamiast

TransponujMacierz (*A*, *n*, *B*);

W Algolu 60 wszystkie ograniczniki parametrów były jednak uważane za równoważne, były więc w istocie komentarzami, a nie częściami nazwy procedury, jak to ma miejsce w Smalltalku.

[11] W Smalltalku wszystko nazywa się inaczej niż gdzie indziej.

3.10. Problemy składni języków

Jakkolwiek wyrażenia są najbardziej różnorodnymi i najciekawszymi elementami języków programowania, problemy składni języków się na nich bynajmniej nie kończą.

3.10.1. Średniki

W językach imperatywnych elementarną jednostką programu jest instrukcja. Instrukcje mogą być ze sobą zestawiane w ciągi instrukcji. Do oddzielania instrukcji od siebie powszechnie używa się średnika. Może on być używany jako:

- *wymagany separator instrukcji* (np. w Pascalu). Służy do oddzielania instrukcji od siebie. Jeżeli za średnikiem nie ma żadnych znaków, to albo przyjmuje się, że za nim znajduje się niejawnie *instrukcja pusta* (tak jest w Pascalu), albo uważa się program za błędny.
- *wymagany terminator instrukcji* (np. w C, C++, Adzie itp.) Dokładnie jeden średnik musi wystąpić na końcu każdej instrukcji.
- *opcjonalny terminator instrukcji* (np. w SML-u, Haskellu itp.) Składnia języka pozwala na jednoznaczny rozbiór programu nawet wówczas, gdy instrukcje nie są rozdzielone średnikiem. Może on być użyty do zwiększenia czytelności programu, lecz można go nie używać wcale.

Pierwsze rozwiązanie wydaje się najgorsze. Jest niezgodne z intuicją wyniesioną z języków naturalnych (kropka jest *terminatorem* a nie *separatorem* zdań w języku polskim), zmiana kolejności dwóch instrukcji wymusza konieczność przeniesienia średnika w inne miejsce, itd. Co prawda można używać średnika po każdej instrukcji, ale powoduje to pojawienie się niejawnych instrukcji pustych. Traktowanie średnika jako separatora jest ponoć źródłem większej liczby błędów w programach, niż traktowanie go jako terminatora. Trzecie rozwiązanie ma natomiast tę wadę, że nie wymusza konsekwentnego zapisu programu. Jest natomiast dosyć bezpieczne i wygodne.

3.10.2. Instrukcje puste

W wielu językach można używać instrukcji pustej „nic nie rób” w miejscach, gdzie musi wystąpić instrukcja, a nie chcemy wykonywania żadnej akcji. Często taka instrukcja jest reprezentowana przez pusty ciąg znaków (np. w Pascalu i C). Instrukcja `while` w C postaci:

```
while (getchar()!='\n');
```

zawiera przed średnikiem instrukcję pustą, podobnie Pascalowa instrukcja złożona

```
begin
   x = x+1;
   y = y-1;
end
```

zawiera przed słowem `end` taką instrukcję. Fakt, że w programie występuje instrukcja, której „nie widać” może być niekiedy przyczyną błędów. Rozważmy dwa programy w C:

```
while (getchar()!='\n');
   i++;
```

oraz

```
while (getchar()!='\n')
   i++;
```

Dopisanie bądź usunięcie średnika istotnie zmienia znaczenie programu. Dlatego w lepszych językach programowania używa się specjalnego słowa kluczowego na oznaczenie instrukcji pustej, np. **skip** w Algolu 68, w którym możemy napisać

while *getchar*() $\neq$ *"*`\n`*"* **do**
  skip*;*

W języku C możemy co prawda napisać

```
while (getchar()!='\n')
   continue;
```

W tym przypadku instrukcja `continue` pełni rolę instrukcji pustej. Problem w języku C polega jednak na tym, że istnieje w nim *możliwość* opuszczenia instrukcji w ciele pętli `while`, podczas gdy w Algolu 68 umieszczenie instrukcji **skip** jest *konieczne*.

3.10.3. Niejednoznaczność składni

Ponieważ fraza `else` w instrukcji warunkowej jest w wielu językach (np. w C) opcjonalna, pojawia się problem jednoznaczności rozbioru programów w zagnieżdżonymi instrukcjami warunkowymi. Dla przykładu gramatyka języka C z książek [82, 83] jest niejednoznaczna:

instrukcja:
  instrukcja-etykietowana
  instrukcja-wyrażeniowa
  instrukcja-złożona
  instrukcja-wyboru
  instrukcja-powtarzania
  instrukcja-skoku
instrukcja-wyboru:
  `if (` *wyrażenie* `)` *instrukcja*
  `if (` *wyrażenie* `)` *instrukcja* `else` *instrukcja*
  `switch (` *wyrażenie* `)` *instrukcja*

i nie określa, jak ma być przeczytana instrukcja:

```
if (x>y) if (x>z) x=1; else x=2;
```

Są dwie możliwości:

```
if (x>y) {
   if (x>z)
      x=1;
}
else x=2;
```

oraz

```
if (x>y) {
   if (x>z)
      x=1;
   else x=2;
}
```

W opisie języka słownie dopowiedziano, że *Niejednoznaczność przyporządkowania* `else` *rozstrzyga się, wiążąc* `else` *z ostatnią instrukcją* `if` *bez* `else`*, napotkaną na tym samym poziomie struktury blokowej* (zatem właściwa jest druga z możliwości przeczytania naszej instrukcji). Taka sama niejednoznaczność występuje w opisie języka C++ [164].

Ponieważ powyższa niejednoznaczność może być źródłem błędów zaleca się, by nie zagnieżdżać instrukcji warunkowych,[12] a jeśli jest to konieczne, dopisywać w odpowiednich miejscach nawiasy `{` i `}`, nawet jeśli reguła wiązania frazy `else` z najbliższym wolnym `if` właściwie rozstrzyga rozbiór danej instrukcji.

Powyższego problemu nie byłoby, gdyby wszystkie instrukcje strukturalne kończyły się jawnym terminatorem. Tak jest w wielu dobrych językach, np. w Algolu 68, w którym instrukcje strukturalne kończy się słowem kluczowym, będącym lustrzanym odbiciem słowa kluczowego rozpoczynającego daną instrukcję, mamy więc

if *warunek* **then** *ciąg-instrukcji* **else** *ciąg-instrukcji* **fi**
while *warunek* **do** *ciąg-instrukcji* **od**
case *wyrażenie* **of** *warianty* **esac**

lub w Adzie, gdzie pisze się **end** i słowo kluczowe rozpoczynające instrukcję, np.

if *warunek* **then** *ciąg-instrukcji* **else** *ciąg-instrukcji* **end if;**
while *warunek* **loop** *ciąg-instrukcji* **end loop;**

W innych językach (np. w Moduli 2) pisze się po prostu **end** na końcu każdej instrukcji, jednak wówczas odnalezienie początku instrukcji zakończonej tym słowem kluczowym może być niekiedy bardziej kłopotliwe. W SML-u słowem `end` kończy się np. wyrażenie `let` i deklaracja `local`. Z konstrukcją `if` nie ma problemu, jest ona bowiem wyrażeniem, nie instrukcją, fraza `else` nie jest w niej zatem opcjonalna. Mimo to w SML-u występują niejednoznaczności rozbioru pewnych konstrukcji podobne do opisanych wyżej (w wyrażeniach

[12] Zagnieżdżenie polega na umieszczeniu instrukcji `if` wewnątrz innej instrukcji `if` *przed* słowem `else`. Nie ma problemu z instrukcjami postaci `if` (e_1) I_1 `else if` (e_2) . . ., gdzie I_1 nie jest instrukcją warunkową.

`case` i `fn`), jednak gramatyka języka podaje reguły rozbioru takich wyrażeń. Zauważmy jednak, że problem nie polega na niejednoznaczności definicji języka (ta bezwzględnie powinna być jednoznaczna; najlepiej gdy jednoznaczna jest sama gramatyka bezkontekstowa opisująca język), tylko na tym, że reguły rozbioru programu nie są oczywiste i intuicyjnie jasne. W SML-u przyjęto np., że `case` wiąże silniej, niż `fn`, zatem

```
fn x => case y of 1 => x | _ => 2
```

znaczy

```
fn x => (case y of 1 => x | _ => 2)
```

a nie

```
fn x => (case y of 1 => x) | _ => 2
```

jednak był to wybór dosyć arbitralny i programiście, który by zapomniał tej reguły trudno było by zgadnąć właściwy sposób rozbioru powyższego wyrażenia. Wymaganie, by wyrażenia `case` i `fn` kończyły się słowem kluczowym `end` usunęłoby ten problem.

3.11. Język D

Jako przykład opisu składni języka programowania definicję języka D, wymyślonego specjalnie na potrzeby niniejszego wykładu. Posłuży nam on również w następnym rozdziale do opisania formalnych metod definiowania semantyki języków programowania.

3.11.1. Składnia języka D

Struktura leksykalna. Tokenami języka D są:

1. *literały całkowitoliczbowe* postaci `[0-9][0-9]*`, tj. niepuste ciągi cyfr dziesiętnych (bez znaku)
2. *słowa kluczowe:* `else false if read skip true while write`
3. *identyfikatory* postaci `[A-Za-z][A-Za-z0-9]*`, tj. ciągi liter i cyfr zaczynające się literą i różne od słów kluczowych
4. *symbole specjalne:*
   (a) *operatory:* `+ - * / % == != <= >= < > ! && ||`
   (b) *symbole pomocnicze:* `= ; ( )`

Wielkie i małe litery są rozróżniane. Białymi znakami są spacje, znaki tabulacji i nowego wiersza oraz komentarze postaci

`/*` *ciąg-znaków* `*/`

gdzie *ciąg-znaków* nie zawiera napisu `*/`. Komentarzy nie można zatem zagnieżdżać. Przy podziale tekstu źródłowego na tokeny obowiązuje zasada zachłanności.

Gramatyka. Gramatyka bezkontekstowa języka D jest przedstawiona w tablicy 3.8, zaś podsumowanie składni wyrażeń — w tablicy 3.9 na stronie 89. Składnia języka D przypomina składnię języka C. Podstawowe różnice polegają na użyciu nawiasów okrągłych `(` i `)` do grupowania instrukcji w miejsce C-owych nawiasów klamrowych `{` i `}` (nawiasy klamrowe przydadzą nam się do innego celu) i rozróżnianie wyrażeń arytmetycznych i logicznych (tak jak w Javie). Wyrażenia arytmetyczne mogą być częścią wyrażeń logicznych i mogą występować w instrukcji przypisania. Wyrażenia logiczne występują w instrukcjach `if` i `while`.

Gramatyka z tablicy 3.8 jest niejednoznaczna z powodu możliwości opuszczania frazy `else` w instrukcji warunkowej. Należy więc dodać słownie, że fraza `else` jest wiązana w drzewie wyprowadzenia z „najbliższym wolnym `if`" (tak jak w C). Gramatyka bezkontekstowa wraz z powyższym słownym opisem tworzą razem jednoznaczny opis składni języka D. Konieczność uzupełniania gramatyki bezkontekstowej słownym opisem jest w pewnej mierze nieelegancka, jednak dzięki temu gramatyka jest zwięzła i czytelna. Jednoznaczna gramatyka języka D wymaga wprowadzenia dodatkowych kategorii składniowych *instrukcja-prosta* i *instrukcja-nasycona* (instrukcja, w której jest tyle słów `else`, co `if`, tj. taka, do której żadna fraza `else` nie może się „dokleić") i zastąpienia produkcji dla kategorii *instrukcja* przez:

instrukcja-prosta	=	`skip;`
	|	`read` identyfikator `;`
	|	`write` *wyrażenie-arytmetyczne* `;`
	|	identyfikator `=` *wyrażenie-arytmetyczne* `;`
	|	`(` *program* `)`
instrukcja-nasycona	=	*instrukcja-prosta*
	|	`if (` *wyrażenie-logiczne* `)` *instrukcja-nasycona* `else` *instrukcja-nasycona*
	|	`while (` *wyrażenie-logiczne* `)` *instrukcja-nasycona*
instrukcja	=	*instrukcja-prosta*
	|	`if (` *wyrażenie-logiczne* `)` *instrukcja*
	|	`if (` *wyrażenie-logiczne* `)` *instrukcja-nasycona* `else` *instrukcja*
	|	`while (` *wyrażenie-logiczne* `)` *instrukcja*

Zauważmy, że gramatyka niejednoznaczna uzupełniona słownym opisem znacznie lepiej zaspokaja potrzeby programisty, który chciałby poznać składnię języka D. Ponadto gramatykę w definicji języka powinno się zapisać w taki sposób, by implementator kompilatora mógł ją użyć do wygenerowania parsera. Parser zwykle generuje się automatycznie, np. programem `ML-Yacc` (pisząc kompilator w SML-u), `yacc` lub `bison` (pisząc kompilator w C). W razie niejednoznaczności gramatyki generator parsera ostrzega o tzw. *konflikcie shift/reduce* i domyślnie wybiera tzw. *reduce*. W przypadku naszej gramatyki będzie to oznaczać właśnie związanie frazy `else` z najbliższym wolnym `if`. Zatem gramatyka niejed-

czynnik	=	identyfikator
	|	literał-całkowitoliczbowy
	|	`-` *czynnik*
	|	`(` *wyrażenie-arytmetyczne* `)`
składnik	=	*czynnik*
	|	*składnik operator-multiplikatywny czynnik*
wyrażenie-arytmetyczne	=	*składnik*
	|	*wyrażenie-arytmetyczne operator-addytywny składnik*
wyrażenie-relacyjne	=	*wyrażenie-arytmetyczne operator-relacyjny wyrażenie-arytmetyczne*
	|	`(` *wyrażenie-logiczne* `)`
	|	`!` *wyrażenie-relacyjne*
	|	`true`
	|	`false`
składnik-logiczny	=	*wyrażenie-relacyjne*
	|	*składnik-logiczny* `&&` *wyrażenie-relacyjne*
wyrażenie-logiczne	=	*składnik-logiczny*
	|	*wyrażenie-logiczne* `||` *składnik-logiczny*
operator-multiplikatywny	=	`*` | `/` | `%`
operator-addytywny	=	`+` | `-`
operator-relacyjny	=	`==` | `!=` | `<=` | `>=` | `<` | `>`
instrukcja	=	`skip;`
	|	`read` identyfikator `;`
	|	`write` *wyrażenie-arytmetyczne* `;`
	|	identyfikator `=` *wyrażenie-arytmetyczne* `;`
	|	`(` *program* `)`
	|	`if (` *wyrażenie-logiczne* `)` *instrukcja*
	|	`if (` *wyrażenie-logiczne* `)` *instrukcja* `else` *instrukcja*
	|	`while (` *wyrażenie-logiczne* `)` *instrukcja*
program	=	*instrukcja*
	|	*program instrukcja*

Tablica 3.8. Gramatyka języka D

operator	priorytet	łączność
− unarny	7	prefiksowy
*, /, %	6	w lewo
+, − binarny	5	w lewo
==, !=, <=, >=, <, >	4	nie są łączne
!	3	prefiksowy
&&	2	w lewo
||	1	w lewo

Tablica 3.9. Operatory w języku D

noznaczna uzupełniona słownymi uwagami jest praktyczniejsza zarówno dla programisty, jak i dla twórcy kompilatora. Jedyny problem, nie związany już bezpośrednio z gramatyką, lecz z definiowanym przez nią językiem polega na tym, że możliwość niejednoznacznego rozbioru programu, sugerowana przez jego budowę gramatyczną, zwiększa ryzyko popełnienia błędu przez programistę. Najlepiej, gdy naturalna, oczywista gramatyka języka od razu jest jednoznaczna. W naszym przypadku było by tak, gdyby instrukcja `if` kończyła się jawnym terminatorem, np. `end` lub `fi`. Skoro jednak postanowiliśmy się wzorować na języku C, konsekwentnie pozostawiamy tę wadę składni języka, mając jednak świadomość jej szkodliwości. Ponieważ większość programistów zna język C i programując w innych językach regularnie popełnia wiele C-izmów,[13] to zmiana składni w tym miejscu mogłaby przynieść więcej szkody niż pożytku.

3.11.2. Nieformalny opis semantyki języka D

Do nazywania komórek pamięci służą w języku D identyfikatory. Komórki pamięci przechowują liczby całkowite. Identyfikatorów (zmiennych) się nie deklaruje, a ich zasięg jest globalny (zatem wszystkie wystąpienia danego identyfikatora w całym programie odnoszą się do tej samej komórki pamięci). Do wczytywania/wypisywania liczb całkowitych służą operacje `read` i `write`.

Przykładowy program w języku D obliczający największy wspólny podzielnik dwóch liczb według algorytmu Euklidesa podanego w tablicy 4.4 na stronie 106 jest przedstawiony w tablicy 3.10.

3.12. Zadania

Zadanie 3.1. Zbuduj automat skończony akceptujący język nad alfabetem $\{0, 1\}$ złożony z tych ciągów zerojedynkowych, które są binarnymi zapisami liczb podzielnych przez 5. Ciąg $a_n \cdots a_0$, gdzie $a_i \in \{0, 1\}$ dla $i = 0, \ldots, n$, jest binarnym zapisem liczby $\sum_{i=0}^{n} 2^i a_i$.

[13] C-izm to użycie konstrukcji języka C w programie napisanym innym języku programowania, podobnie jak np. rusycyzm to użycie zwrotów rosyjskich w języku polskim.

```
read X;
read Y;
while (Y != 0) (
   Z = X % Y;
   X = Y;
   Y = Z;
)
write X;
```

Tablica 3.10. Implementacja algorytmu Euklidesa obliczania największego wspólnego podzielnika w języku D

Zadanie 3.2. Rozważmy gramatykę nad alfabetem terminalnym $\Sigma = \{0, 1, (,), +\}$ z jednym symbolem nieterminalnym E i zbiorem produkcji

$$P \quad = \quad \{E \to 0, E \to 1, E \to (E), E \to E{+}E\}$$

Napisz wszystkie możliwe drzewa wyprowadzenia i po jednym przykładowym wyprowadzeniu słów:

1. 1+1+(0+1+1)
2. 1+0+1
3. (1+0)+1
4. 1+(0+1)

Zadanie 3.3. Zdefiniuj gramatykę bezkontekstową generującą język wszystkich *palindromów* nad alfabetem {a, b}. Słowo jest palindromem, jeśli przeczytane wspak jest sobie równe. *Informacja:* nie istnieje gramatyka regularna (typu 3) generująca ten język.

Zadanie 3.4. Zdefiniuj gramatykę typu 0 generującą język $L = \{ww \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$, tj. zbiór wszystkich słów będących dwukrotnym powtórzeniem jakiegoś słowa. *Informacja:* faktycznie wystarczy tu gramatyka kontekstowa (typu 1), nie istnieje natomiast gramatyka bezkontekstowa (typu 2) generująca ten język.

Zadanie 3.5. Słowo x nad alfabetem $\{(,)\}$ jest *ciągiem poprawnie rozstawionych nawiasów*, jeśli bilans$(x) = 0$ i bilans$(y) \geq 0$, dla każdego prefiksu y słowa x, gdzie bilans$(\epsilon) = 0$, bilans$(z() =$ bilans$(z) + 1$, oraz bilans$(z)) =$ bilans$(z) - 1$. Dla przykładu (((()))()) jest ciągiem poprawnie rozstawionych nawiasów, zaś ()))(() nim nie jest. Udowodnij, że gramatyka $G = \langle\{(,)\}, \{S\}, S, \{S \to (S), S \to SS, S \to \epsilon\}\rangle$ generuje zbiór wszystkich ciągów poprawnie rozstawionych nawiasów.

Zadanie 3.6. Niech $|w|_0$ i $|w|_1$ oznaczają liczbę wystąpień odpowiednio zer i jedynek w słowie $w \in \{0, 1\}^*$. Napisz gramatykę generującą język

$$L \quad = \quad \{w \in \{0, 1\}^* \ : \ |w|_0 = 2|w|_1 \wedge \forall v \leq w.|w|_0 \leq 2|w|_1\}$$

gdzie $v \leq w$ oznacza, że v jest prefiksem słowa w. *Wskazówka:* należy wzorować się na poprzednim zadaniu, przy czy teraz definiujemy $\text{bilans}(\epsilon) = 0$, $\text{bilans}(z0) = \text{bilans}(z) - 1$ i $\text{bilans}(z1) = \text{bilans}(z) + 2$.

Zadanie 3.7. Niech $\Sigma = \{\mathtt{a}, \mathtt{b}\}$,

$$\begin{aligned} \text{bilans}(\epsilon) &= 0 \\ \text{bilans}(w\mathtt{a}) &= \text{bilans}(w) + 1 \\ \text{bilans}(w\mathtt{b}) &= \text{bilans}(w) - 1 \end{aligned}$$

dla $w \in \Sigma^*$ i

$$\begin{aligned} L &= \{w \in \Sigma^* \mid \forall v \leq w.\text{bilans}(v) \geq 0\} \\ G_1 &= \langle \Sigma, \{S\}, S, \{S \to \epsilon, S \to \mathtt{a}S\mathtt{b}S, S \to \mathtt{a}S\} \rangle \\ G_2 &= \langle \Sigma, \{N\}, N, \{N \to \epsilon, N \to \mathtt{a}N\mathtt{b}N\} \rangle \\ G_3 &= \langle \Sigma, \{N, T\}, T, \{N \to \epsilon, N \to \mathtt{a}N\mathtt{b}N, T \to \epsilon, T \to \mathtt{a}N\mathtt{b}T, T \to \mathtt{a}T\} \rangle \end{aligned}$$

gdzie $v \leq w$ oznacza, że słowo v jest prefiksem słowa w. Wzorując się na zadaniach poprzednich udowodnij, że:

1. $\mathfrak{L}(G_1) = L$,
2. $\mathfrak{L}(G_3) = L$,
3. $\mathfrak{L}(G_2) = L \cap \{w \in \Sigma^* \mid \text{bilans}(w) = 0\}$,
4. G_1 nie jest jednoznaczna,
5. G_2 jest jednoznaczna,
6. G_3 jest jednoznaczna (*wskazówka*: rozważ dwa drzewa wyprowadzenia pewnego słowa; pokaż że ich korzenie muszą być takie same; dalej przez indukcję względem struktury drzew pokaż, że drzewa te są równe).

Wniosek: $\mathfrak{L}(G_1) = \mathfrak{L}(G_3)$ i G_3 jest jednoznaczna.

Zadanie 3.8. Opcjonalna fraza `else` w gramatyce

$$\begin{aligned} S \quad ::= \quad & \texttt{if}\ E\ \texttt{then}\ S\ \texttt{else}\ S \\ \mid \quad & \texttt{if}\ E\ \texttt{then}\ S \\ \mid \quad & I \end{aligned}$$

gdzie I jest instrukcją prostą powoduje, że gramatyka ta jest niejednoznaczna. Korzystając z wyników poprzedniego zadania napisz jednoznaczną gramatykę opisującą ten sam język, w której fraza `else` jest wiązana z najbliższym „wolnym” `if`. Uzasadnij, że wersja gramatyki języka D podana na stronie 87 jest jednoznaczna i opisuje ten sam język, co gramatyka podana w tablicy 3.8 na stronie 88.

Zadanie 3.9. Udowodnij, że gramatyka opisana w podrozdziale 3.2.3 opisuje zbiór wyrażeń zbudowanych z wyrażeń atomowych, operatorów $\oplus_1, \ldots \oplus_n$ i nawiasów, jest jednoznaczna i zgodna z przyjętymi priorytetami i łączliwością operatorów.

Zadanie 3.10. Pokaż, że każdy język bezkontekstowy nie zawierający słowa pustego można opisać gramatyką bezkontekstową w tzw. *postaci Chomsky'ego*, w której wszystkie produkcje są postaci:

$$X \to YZ \qquad \text{lub} \qquad X \to a$$

gdzie X, Y i Z są symbolami nieterminalnymi, zaś a jest symbolem terminalnym.

Zadanie 3.11. Zaprojektuj algorytm i zaimplementuj go w Pascalu, C lub SML-u, który dla zadanej gramatyki w postaci Chomsky'ego sprawdza, czy podane słowo $a_1 \ldots a_n$ należy do języka opisanego tą gramatyką. Wykorzystaj w tym celu technikę *programowania dynamicznego* (zob. [6], str. 92–95). Algorytm powinien wyznaczyć dla każdego podsłowa $a_i \ldots a_j$ ($1 \le i \le j \le n$) zbiór symboli nieterminalnych, z których daje się to słowo wyprowadzić. Jest to łatwe dla podsłów długości 1. Dalej należy postępować indukcyjnie. Słowo należy do języka, jeśli symbol startowy gramatyki należy do zbioru symboli, z których dane słowo daje się wyprowadzić.

Zadanie 3.12. Napisz gramatykę bezkontekstową generującą język nad alfabetem

$$\{(,), \vee, \wedge, \Rightarrow, \Leftrightarrow, \neg, \top, \bot\}$$

złożony z formuł rachunku zdań nie zawierających zmiennych zdaniowych, gdzie $\top$ oznacza prawdę, $\bot$ fałsz, a priorytety operatorów są następujące:

operator	priorytet	kierunek łączności
$\neg$	5	prefiksowy
$\wedge$	4	w lewo
$\vee$	3	w lewo
$\Rightarrow$	2	w prawo
$\Leftrightarrow$	1	w lewo

Przepisz tę gramatykę w notacji BNF, EBNF, w notacji z definicji języka C i narysuj odpowiednie diagramy syntaktyczne.

Zadanie 3.13. Na wzór programu z podrozdziału 3.4 napisz w C funkcję

```
bool formuła (char *napis, bool *wartość);
```

lub napisz w Pascalu funkcję

```
function formuła (napis : string; var wartość : boolean) : boolean;
```

sprawdzającą, czy jej argument zawiera poprawnie zapisaną formułę zdaniową z poprzedniego zadania (alfabet reprezentujemy za pomocą następujących napisów ASCII: `(`, `)`, `|`, `&`, `=>`, `<=>`, `~`, `F`, `T`) i jeśli tak, wyznaczającą wartość logiczną tej formuły.

Zadanie 3.14. *Literał zmiennopozycyjny* w Pascalu to napis nad alfabetem

$$\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{+}, \texttt{-}, \texttt{e}, \texttt{.}\}$$

następującej postaci: na początku występuje opcjonalnie znak `+` lub `-`, następnie niepusty ciąg cyfr `0`...`9`, potem opcjonalnie część ułamkowa, tj. kropka i niepusty ciąg cyfr i na końcu opcjonalnie wykładnik, tj. znak `e`, opcjonalnie znak `+` lub `-` i niepusty ciąg cyfr. Co najmniej jeden napis spośród części ułamkowej i wykładnika musi wystąpić (inaczej literał będzie tzw. literałem całkowitoliczbowym). Napisz gramatykę regularną generującą zbiór literałów zmiennopozycyjnych w Pascalu.

Zadanie 3.15. Podaj formalny opis algorytmu przekształcania gramatyki EBNF do BNF, zgodnie z ideą opisaną w podrozdziale 3.3.3 i udowodnij jego poprawność (tj. pokaż, że zawsze się zatrzymuje i otrzymana gramatyka opisuje ten sam język, co wyjściowa).

Zadanie 3.16. Notację EBNF rozszerzamy dodatkowo o zapis postaci $\{w\}_n^m$, który dla $0 \leq n \leq m$ oznacza co najmniej n i co najwyżej m powtórzeń w (przypomnijmy, że w oryginalnym EBNF zapis $\{w\}$ oznacza dowolnie wiele, co najmniej jedno powtórzenie w), np. gramatyka

liczba $=$ $\{$*cyfra*$\}_1^6$
cyfra $=$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

opisuje zbiór liczb co najwyżej sześciocyfrowych. Pokaż że jest to jedynie udogodnienie skracające zapis, które nie zwiększa klasy języków opisywanych przez gramatykę, tj. pokaż, jak dowolną gramatykę w rozszerzonej notacji przekształcić do oryginalnej EBNF. Pokaż że w tej notacji nawiasy kwadratowe $[\cdot]$ są zbyteczne. Korzystając z powyższego udogodnienia zapisz gramatykę opisującą napisy na starych polskich tablicach rejestracyjnych (tych z czarnym tłem). Prawda że się przydaje?

Zadanie 3.17. Napisz program (w C lub Pascalu), który jako parametr podany w wierszu poleceń podczas uruchamiania programu otrzymuje napis (wzorzec) o długości co najwyżej 256 znaków i który wczytuje tekst (dowolnej długości) ze standardowego strumienia wejściowego a następnie wypisuje liczbę wystąpień podanego wzorca w tym tekście. Program powinien implementować algorytm wyszukiwania z użyciem automatu opisany w podrozdziale 3.7.3.

Zadanie 3.18. Napisz wyrażenia regularne opisujące poniższe języki nad alfabetem $\Sigma = \{\texttt{a}, \texttt{b}\}$:

1. słowa w których każdym prefiksie różnica liczby symboli `a` i `b` jest nie większa niż jeden,
2. słowa w których każdym prefiksie różnica liczby symboli `a` i `b` jest nie większa od dwóch,
3. słowa nie zawierające ciągu `bb`.

Zadanie 3.19. Pokaż że jeśli dla pewnego języka istnieje automat skończony, który go rozpoznaje, to język ten jest regularny, tj. istnieje odpowiednia gramatyka regularna, która go generuje. *Informacja:* prawdziwe jest również twierdzenie odwrotne. Czy dowód tego twierdzenia jest trudniejszy?

Zadanie 3.20. Wszystkie operatory binarne w Pascalu łączą w lewo i są podzielone na cztery grupy pod względem priorytetu (od najmniejszego do największego):

```
< <= = <> >= > in
+ - or
* / div mod and
not
```

Dodatkowo unarny operator `-` może pojawić się przed liczbą i nawiasem otwierającym, ale nie po operatorze binarnym, np. `-a` oraz `-(1+2)` są poprawne, `a+-b` zaś — nie. Napisz gramatykę bezkontekstową w notacji EBNF opisującą wyrażenia w Pascalu.

Zadanie 3.21. Dla każdego z poniższych wyrażeń w Pascalu narysuj drzewo wyprowadzenia tego wyrażenia z gramatyki z poprzedniego zadania oraz abstrakcyjne drzewo rozbioru tego wyrażenia.

1. `i >= 0`
2. `(i >= 0) and not p`
3. `i >= 0 and not p`
4. `(i>= 0) and (x <> y)`

Zadanie 3.22. Napisz w oryginalnej notacji BNF (z użyciem nawiasów kątowych do oznaczania symboli nieterminalnych i bez rozszerzeń z notacji EBNF) jednoznaczną gramatykę opisującą wyrażenia złożone z literałów całkowitoliczbowych, identyfikatorów, nawiasów i następujących operatorów binarnych:

operator	kierunek łączności	priorytet
^	w prawo	4
*	w lewo	3
+	w lewo	2
<	nie jest łączny	1
=	nie jest łączny	1

Zadanie 3.23. Zamień na odwrotną notację polską i narysuj abstrakcyjne drzewa rozbioru następujących wyrażeń:

1. `(23+45)*(53+27)`
2. `1+2+3*4*5`

3. `12-7*(13+5)*4`
4. `56+23*45-12*23`
5. `45+(60-34)-12`
6. `23+(56+67)/23`
7. `49*56^2+23*17^2^3`
8. `37+45^4+12*23`
9. `27-(12+45*(13+2)^2)`
10. `(14+56)+12+(13+3)`

Oblicz wartość poniższych wyrażeń i zapisz je w notacji infiksowej:

1. `37 12 + 45 3 - 2 ^ *`
2. `3 4 2 + * 4 ^ 2 * 12 3 5 - * -`
3. `56 4 12 2 / 3 / * + 2 -`
4. `34 56 + 2 2 2 ^ ^ * 12 67 + *`
5. `23 45 12 * + 15 -`
6. `56 67 * 26 - 12 23 2 ^ * +`
7. `37 45 * 12 48 + * 2 3 ^ +`
8. `37 56 * 76 * 58 + 12 56 * -`
9. `34 45 + 34 27 + *`
10. `49 45 12 - 67 45 * * 34 2 / 2 ^ * +`

Zadanie 3.24. Rozbuduj program z podrozdziału 3.6.5 przekształcający wyrażenia na odwrotną notację polską tak, by akceptował dowolne wyrażenia języka C i wykrywał wszystkie błędy składniowe we wczytywanych wyrażeniach. Zauważ, że w C są również operatory miksfiksowe, np. `?:`. Wyrażenie e_1 `?` e_2 `:` e_3 powinno być przetłumaczone na $e'_1\ e'_2\ e'_3$ `?`, gdzie e'_1, e'_2, e'_3 są tłumaczeniami wyrażeń e_1, e_2, e_3.

Zadanie 3.25. Wyrażenia regularne e_1 i e_2 *są równe* ($e_1 = e_2$), jeżeli $\mathfrak{L}(e_1) = \mathfrak{L}(e_2)$. Udowodnij, że dla dowolnych wyrażeń e_1, e_2 i e_3 prawdziwe są następujące równości:

1. $e_1 + e_2 = e_2 + e_1$ ($+$ jest przemienny)
2. $(e_1 + e_2) + e_3 = e_1 + (e_2 + e_3)$ ($+$ jest łączny)
3. $(e_1e_2)e_3 = e_1(e_2e_3)$ (konkatenacja jest łączna)
4. $(e_1 + e_2)e_3 = e_1e_3 + e_2e_3$ i $e_1(e_2 + e_3) = e_1e_2 + e_1e_3$ (prawa rozdzielności konkatenacji względem $+$)
5. $\emptyset + e_1 = e_1$ i $e_1 + \emptyset = e_1$ ($\emptyset$ jest obustronnym elementem neutralnym $+$)
6. $e_1\epsilon = e_1$ i $\epsilon e_1 = e_1$ (ϵ jest obustronnym elementem neutralnym konkatenacji)

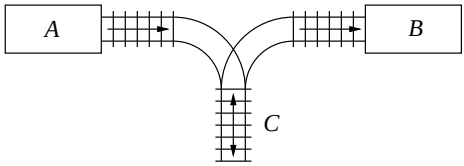
7. $\emptyset^* = \{\epsilon\}$
8. $(e_1^*)^* = e_1^*$ (* jest idempotentna)
9. $(e_1^* e_2^*)^* = (e_1 + e_2)^*$
10. $(e_1 + \epsilon)^* = e_1^*$ i $(\epsilon + e_1)^*$

Zadanie 3.26. Na wzór stosów omówionych w podrozdziale 3.6.1 napisz plik `queue.c` implementujący kolejki zgodnie z plikiem nagłówkowym `queue.h` o treści:

```
/* queue.h: Specyfikacja kolejek liczb calkowitych */

#define QUEUE_SIZE 200

typedef struct {
   int front, rear;
   int tbl [QUEUE_SIZE];
} queue;

void init (queue *q);
int front (queue *q, int *x);
int put (queue *q, int x);
int get (queue *q, int *x);
int rem (queue *q);
int empty (queue *q);
```

Kolejka jest strukturą o dostępie sekwencyjnym typu FIFO (*first in, first out*). Elementy wchodzą do kolejki z jednej strony, a wychodzą z drugiej. Element, który wszedł do kolejki jako pierwszy, najwcześniej ją opuści. Funkcja `put` wstawia element do kolejki, `front` ujawnia element, który jest na przedzie kolejki (najwcześniej do niej wszedł), `get` — wyjmuje ten element z kolejki, `rem` — usuwa ten element z kolejki, `empty` — sprawdza, czy kolejka jest pusta, zaś `init` sprawia, że kolejka staje się pusta. Funkcje `first`, `get` i `rem` zwracają `0`, jeśli kolejka była niepusta i `1` w przeciwnym razie, funkcja `put` zwraca `1` w razie przepełnienia kolejki i `0` jeśli wstawianie się powiodło. W implementacji kolejki zmienne `front` i `rear` wskazują odpowiednio na pierwszy zajęty i pierwszy wolny element tablicy.

Zadanie 3.27. Dane są dwie stacje kolejowe A i B, jak na rysunku 3.8. Na stacji A stoi pociąg z wagonami ponumerowanymi kolejnymi liczbami od 1 do n. Chcemy przetoczyć wagony do stacji B, tak, aby były ustawione w pewnej kolejności. Napisz program, który dla zadanej kolejności wagonów na stacji B odpowiada, czy jest to możliwe. Wagony można przetaczać grupami, spinać i rozpinać w dowolnym miejscu, jednak mogą się one poruszać tylko w jednym kierunku: ze stacji A do zwrotnicy C i ze zwrotnicy C do stacji B. Z powrotem ich przetaczać nie wolno. Wagonów nie wolno też wykoleić! Zauważ że zwrotnica C jest stosem (wykorzystaj go w programie).



Rysunek 3.8. Tory kolejowe łączące stacje *A* i *B* i zwrotnicę *C*

Zadanie 3.28. Napisz w języku D programy opisane w zadaniu 4.1.

Zadanie 3.29. Napisz parser języka D, tj. program, który będzie dokonywał rozbioru gramatycznego programów w języku D i wskazywał występujące w nich błędy składniowe (por. zadanie 6.1).

Rozdział 4

Programowanie niskopoziomowe

W niniejszym rozdziale opisujemy, jak działa komputer i zajmujemy się językami programowania najniższego poziomu — językiem maszynowym i asemblerem. Ponieważ procesory znajdujące się wewnątrz współczesnych komputerów są zbyt skomplikowane by na ich przykładzie przedstawić zasadę działania komputera, poniżej opisujemy procesor Sextium II, wymyślony specjalnie na potrzeby naszego wykładu.

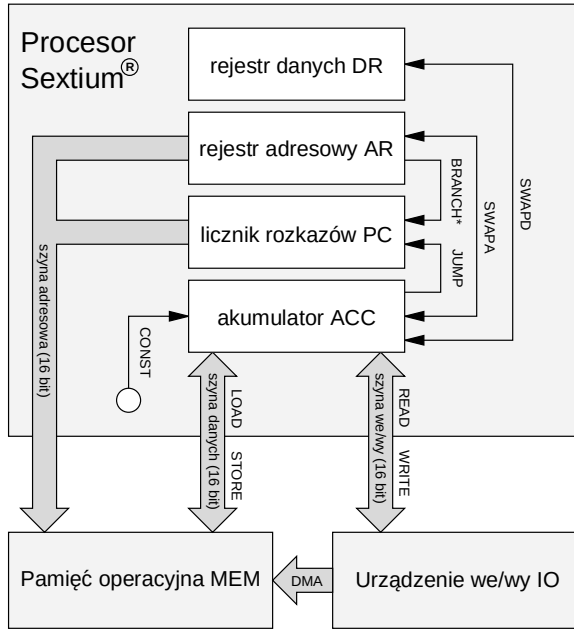
4.1. Opis procesora Sextium II

Budowa procesora Sextium II[1] o architekturze typu RISC[2] jest przedstawiona na rysunku 4.1. Poprzez szynę danych i szynę adresową do procesora jest podłączona pamięć (MEM) złożona z 64k słów 16-bitowych, a poprzez szynę wejścia/wyjścia — urządzenie wejścia/wyjścia (IO). Szyny: danych, adresowa i we/wy są 16-bitowe, podobnie jak rejestry: akumulator (ACC), rejestr danych (DR), rejestr adresowy (AR) i licznik rozkazów (PC). Rozkazy (jest ich 16) zajmują po 4 bity i są pakowane po 4 w słowie maszyny. Opis rozkazów jest przedstawiony w tablicach 4.1 i 4.2, a sposób pakowania rozkazów w słowie na rysunku 4.2.

Do przedstawiania zawartości pamięci i rejestrów procesora będziemy konsekwentnie używać zapisu szesnastkowego. Zawartość pojedynczego słowa będzie zatem opisana cią-

[1] Pierwsza wersja procesora Sextium, o czym przekonali się studenci rocznika 1999/2000, była nieudana i została wkrótce zastąpiona modelem Sextium II.

[2] *Reduced instruction set computer*, procesor o ograniczonym zbiorze instrukcji: procesor w którym lista rozkazów jest maksymalnie skrócona. Dzięki temu rozkazy mogą być wykonywane bardzo szybko i sprawnie. Procesory RISC (posiadające kilkanaście do kilkudziesięciu rozkazów) są przy tej samej częstotliwości zegara kilkakrotnie bardziej efektywne niż procesory CISC (*complex instruction set computer*). Wynika to z faktu, że jeden cykl rozkazowy w takim procesorze trwa tylko jeden lub najwyżej kilka taktów zegara a nie kilkanaście czy kilkadziesiąt jak w procesorach typu CISC. Rozbudowana lista rozkazów może pomóc zoptymalizować kod wynikowy, jeśli programista pisze program w asemblerze, natomiast zbudowanie kompilatora języka wysokiego poziomu, który wykorzystywałby wiele rozkazów jest kłopotliwe. W praktyce kompilatory używają tylko najprostszych rozkazów procesora CISC i dodatkowe możliwości takiego procesora związane z bardzo bogatą listą instrukcji nie są wykorzystywane (optymalizacja kodu jest trudna). Tworzenie kompilatorów generujących efektywny kod dla procesorów RISC jest łatwiejsze niż dla procesorów CISC (efektywność jest osiągnięta dzięki masowości a nie różnorodności).



ACC, *accumulator*, akumulator: rejestr procesora, na którym są wykonywane operacje arytmetyczne i przesyłu danych.

DR, *data register*, rejestr danych: pomocniczy rejestr przechowujący drugi argument operacji arytmetycznych.

AR, *address register*, rejestr adresowy: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania (zapisu) danych z (do) pamięci.

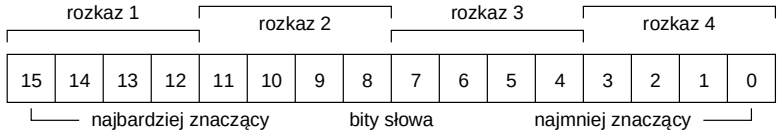
PC, *program counter*, licznik rozkazów: rejestr, którego zawartość służy do adresowania pamięci w celu pobrania kolejnych rozkazów do wykonania.

MEM, *memory*, pamięć operacyjna.

IO, *input/output*, urządzenie wejścia/wyjścia.

DMA, *direct memory access*, możliwość bezpośredniego zapisu do pamięci wprost z urządzenia wejścia/wyjścia bez udziału procesora.

Rysunek 4.1. Architektura procesora Sextium II



Rysunek 4.2. Sposób pakowania rozkazów procesora Sextium II w słowie maszynowym

giem czterech cyfr szesnastkowych. Dla wygody rozkazy mają swoje nazwy mnemoniczne. Zatem np. zamiast pisać „rozkaz `E`", będziemy pisać „rozkaz `DIV`". Dla procesora jednak rozkaz, to po prostu cztery bity (które my zapisujemy za pomocą pojedynczej cyfry szesnastkowej).

Cykl rozkazowy procesora polega na pobieraniu, dekodowaniu i wykonywaniu rozkazów. Wczytane słowo zawierające cztery rozkazy jest przechowywane w specjalnym *buforze rozkazów* (nie uwidocznionym na rysunku 4.1). W celu pobrania rozkazów procesor odwołuje się do pamięci przeważnie raz na cztery cykle rozkazowe, chyba że wykonuje instrukcję skoku. Wówczas bufor rozkazów jest opróżniany. Można zatem skoczyć jedynie do rozkazu, który zajmuje cztery najbardziej znaczące bity słowa (jeśli sprawia to jakieś problemy, można poprzednie słowo wypełnić rozkazami `NOP` w celu wyrównania wskazanego rozkazu do granicy słowa). Rozkaz `CONST` służy do załadowania stałej do akumulatora. Stała jest umieszczana w słowie następującym bezpośrednio po słowie zawierającym ten rozkaz (lub w następnych słowach, jeżeli w jednym słowie znajduje się więcej niż jeden rozkaz `CONST`). Np. następująca para słów: `6AB0 0001` zawiera ciąg rozkazów `SWAPD`, `CONST`, `ADD`, `NOP`, po którym następuje liczba 1. Wykonanie tych rozkazów powoduje zwiększenie zawartości akumulatora o jeden. Podobnie ciąg sześciu słów:

```
A6AD 0004 0005 6A56 FFFA 2000
```

który symbolicznie zapiszemy jako:

```
CONST SWAPD CONST MUL
0004
0005
SWAPD CONST SWAPA SWAPD
FFFA
STORE NOP NOP NOP
```

powoduje załadowanie do rejestru danych liczby 4, do akumulatora liczby 5, przemnożenie zawartości akumulatora przez zawartość rejestru danych i zapisanie wyniku (liczby 20) do komórki pamięci o adresie `FFFA`.

Program dla procesora jest wpisywany do pamięci wprost z urządzenia wejścia/wyjścia (poprzez tzw. szynę DMA). W tym czasie procesor nie pracuje. Następnie urządzenie wejścia/wyjścia wysyła sygnał do procesora, który zeruje zawartość wszystkich swoich rejestrów i opróżnia bufor rozkazów, a następnie rozpoczyna wykonywanie cyklu rozkazowego.

rozkaz	symb.	opis	uwagi
0	NOP	nic nie rób	przejdź do wykonania następnego rozkazu
1	LOAD	MEM[AR] $\rightarrow$ ACC	załaduj słowo o adresie znajdującym się w rejestrze adresowym do akumulatora
2	STORE	ACC $\rightarrow$ MEM[AR]	zapisz zawartość akumulatora do słowa o adresie znajdującym się w rejestrze adresowym
3	READ	IO $\rightarrow$ ACC	wczytaj słowo z urządzenia wejściowego do akumulatora
4	WRITE	ACC $\rightarrow$ IO	prześlij zawartość akumulatora do urządzenia wyjściowego
5	SWAPA	ACC $\leftrightarrow$ AR	zamień miejscami zawartość rejestru adresowego i akumulatora
6	SWAPD	ACC $\leftrightarrow$ DR	zamień miejscami zawartość rejestru danych i akumulatora
7	JUMP	ACC $\rightarrow$ PC	wpisz do licznika instrukcji zawartość akumulatora (skocz do instrukcji o adresie znajdującym się w akumulatorze)
8	BRANCHZ	if ACC $= 0$ then AR $\rightarrow$ PC	jeżeli akumulator zawiera liczbę 0, zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)
9	BRANCHP	if ACC > 0 then AR $\rightarrow$ PC	jeżeli akumulator zawiera liczbę dodatnią, zapisz do licznika instrukcji zawartość rejestru adresowego (skocz do instrukcji o adresie znajdującym się w rejestrze adresowym)

Tablica 4.1. Zestaw rozkazów procesora Sextium II (część 1)

rozkaz	symb.	opis	uwagi
A	CONST	MEM[PC++] → ACC	zapisz słowo znajdujące się w komórce pamięci o adresie wskazywanym przez bieżącą zawartość licznika rozkazów do akumulatora
B	ADD	ACC + DR → ACC	dodaj do akumulatora zawartość rejestru danych
C	SUB	ACC − DR → ACC	odejmij od akumulatora zawartość rejestru danych
D	MUL	ACC × DR → ACC	pomnóż zawartość akumulatora przez zawartość rejestru danych
E	DIV	ACC / DR → ACC	podziel zawartość akumulatora przez zawartość rejestru danych
F	HALT	zatrzymaj	przekaż sterowanie do urządzenia we/wy

Tablica 4.2. Zestaw rozkazów procesora Sextium II (dokończenie)

Procesor przekazuje sterowanie na powrót do urządzenia wejścia/wyjścia po wykonaniu rozkazu HALT lub w razie próby dzielenia przez zero. Pracę procesora może również przerwać urządzenie wejścia/wyjścia. Po uruchomieniu licznik rozkazów ma wartość 0000, zatem procesor rozpoczyna działanie od wykonania rozkazów znajdujących się w zerowym słowie pamięci. Algorytm pracy procesora jest opisany w tablicy 4.3, w której A++ oznacza zwiększenie zawartości rejestru A o jeden.

Słowa w operacjach arytmetycznych reprezentują liczby całkowite ze znakiem z przedziału $-2^{15} \div (2^{15} - 1)$ w zapisie uzupełnieniowym do dwóch, tj. zmiana znaku liczby polega na zanegowaniu wszystkich bitów liczby i dodaniu jedynki. Słowa od 0000 do 7FFF reprezentują liczby nieujemne 0 do 32767, słowa 8000 do FFFF zaś liczby -32768 do -1.

4.2. Asembler

Zapisywanie programu wprost w kodzie maszynowym jest kłopotliwe. Asembler, inaczej *język adresów symbolicznych*, pozwala na symboliczne zapisywanie rozkazów i adresów uwalniając programistę od żmudnego i mechanicznego kodowania programu. Jest to przy tym język niskiego poziomu, w którym programista ma pełną kontrolę nad postacią kodu

```
wyzeruj rejestry ACC, PC, AR i DR i opróżnij bufor rozkazów
loop
  if bufor rozkazów jest pusty then
    pobierz słowo z pamięci o adresie zawartym w PC
    PC++
  end if
  R ← odkoduj następny rozkaz
  if R = LOAD, STORE, READ, WRITE, SWAPA, SWAPD, ADD, SUB, MUL, DIV then
    wykonaj tę operację zgodnie z opisem w tablicach 4.1 i 4.2
  else if R = CONST then
    załaduj słowo z pamięci o adresie zawartym w PC do ACC
    PC++
  else if R = JUMP then
    zapisz do PC zawartość ACC
    opróżnij bufor rozkazów
  else if (R = BRANCHZ i ACC = 0) lub (R = BRANCHP i ACC > 0) then
    zapisz do PC zawartość AR
    opróżnij bufor rozkazów
  else if R = HALT then
    zatrzymaj pracę
  end if
end loop
```

Tablica 4.3. Cykl rozkazowy procesora Sextium II

wynikowego. Zwykle jedna instrukcja asemblera jest przekładana na jeden rozkaz maszyny. Program w asemblerze jest dokładnym opisem kodu maszynowego, w którym rozkazy są reprezentowane przez ich nazwy mnemoniczne a adresy przez identyfikatory. Program tłumaczący (również zwany asemblerem) składa (ang. *assemble*) kod maszynowy według tego opisu, wstawiając kody rozkazów w miejsce ich nazw mnemonicznych, a wyliczone adresy w miejsce identyfikatorów. W językach wysokiego poziomu, takich jak np. SML, progamista posługuje się zwykle pojęciem *maszyny abstrakcyjnej*, której budowa jest nieraz bardzo odległa od architektury rzeczywistego procesora, na którym jest wykonywany skompilowany program (np. w SML-u dołączenie głowy do listy jest traktowane jako pojedyncza operacja, tymczasem przekłada się ono na ciąg wielu rozkazów procesora). Natomiast programując w asemblerze programista nadal myśli w kategoriach rzeczywistego procesora. Ponieważ asembler jest ściśle związany z architekturą danego procesora, w odróżnieniu od języków wysokiego poziomu, asembler jest zależny od maszyny. Dlatego nie istnieje jeden język zwany asemblerem, są tylko asemblery konkretnych procesorów. Aby język programowania był niezależny od procesora, jego maszyna abstrakcyjna musi być idealizacją, częścią wspólną wszystkich procesorów, na które programy w tym języku mają być kompilowane. Przykładem takiego języka jest C, zwany niekiedy *asemblerem strukturalnym*. Należy on już

jednak do kolejnego piętra w hierarchii „poziomu” języków programowania.

4.2.1. Asembler procesora Sextium II

Jednostkami leksykalnymi asemblera procesora Sextium II są *literały całkowitoliczbowe*, tj. niepuste ciągi cyfr dziesiętnych, poprzedzone opcjonalnie znakiem minus, np. `73`, *literały szesnastkowe*, tj. ciągi dokładnie czterech cyfr szesnastkowych poprzedzone znakami `0x` lub `0X`, np. `0xFAFA`, *słowa kluczowe*:

```
ADD BRANCHP BRANCHZ CONST DATA DIV HALT JUMP LOAD MUL READ STORE SUB
SWAPA SWAPD WRITE
```

identyfikatory, tj. ciągi liter i cyfr zaczynające się literą i różne od słów kluczowych oraz *symbol pomocniczy* „`:`”. Identyfikatory są używane jako *etykiety*. Wielkie i małe litery są rozróżniane w identyfikatorach i słowach kluczowych, natomiast cyfry szesnastkowe `A`–`F` można pisać zarówno wielką, jak i małą literą. Znakiem początku komentarza jest `#`. Komentarz rozciąga się do końca wiersza w którym występuje (do znaku nowego wiersza).

Program w asemblerze jest ciągiem *instrukcji asemblera*, po co najwyżej jednej w wierszu. Instrukcja może się składać z pojedynczego słowa kluczowego

```
ADD BRANCHP BRANCHZ DATA DIV HALT JUMP LOAD MUL READ STORE SUB SWAPA
SWAPD WRITE
```

lub może być postaci

`CONST` *parametr*

gdzie *parametr* to albo literał dziesiętny lub szesnastkowy, albo identyfikator (etykieta). Każda instrukcja może być opcjonalnie poprzedzona napisem postaci

etykieta `:`

Zauważmy że rozkaz procesora `NOP` *nie jest* instrukcją asemblera, natomiast instrukcja `DATA` nie ma odpowiednika wśród rozkazów procesora. Identyfikatory (etykiety) są symbolicznymi reprezentacjami adresów pamięci. Każda etykieta powinna pojawić się dokładnie raz w programie przed dwukropkiem i może pojawiać się wielokrotnie w instrukcji `CONST`. Zadaniem asemblera jest przetłumaczenie symbolicznych nazw instrukcji na odpowiadające im rozkazy, spakowanie rozkazów po cztery w 16-bitowe słowa (wypełniając puste miejsca rozkazami `NOP`), wyznaczenie faktycznych adresów instrukcji opatrzonych etykietami i wygenerowanie kodu binarnego. Asembler generuje rozkazy w takiej kolejności, w jakiej odpowiednie instrukcje znajdują się w tekście programu. Instrukcja `CONST` powoduje wygenerowanie rozkazu ładującego stałą opisaną podanym literałem lub etykietą do akumulatora. Instrukcja `DATA` powoduje zarezerwowanie słowa w pamięci w miejscu, w którym występuje (dlatego zwykle umieszcza się ją na końcu programu). Do tego słowa w pamięci wpisywana jest liczba `0x0000` (tj. pamięć dla zmiennych jest inicjowana podczas uruchamiania programu). Do adresu tego słowa można się odwoływać poprzez etykietę tej instrukcji (zatem użycie rozkazu `DATA` bez etykiety ma niewiele sensu). Program tłumaczący program w

wczytaj x
wczytaj y
while $y \neq 0$ **do**
  $z \leftarrow x \bmod y$
  $x \leftarrow y$
  $y \leftarrow z$
end while
wypisz x

Tablica 4.4. Algorytm Euklidesa obliczania największego wspólnego podzielnika

asemblerze na kod wynikowy jest dwuprzebiegowy: w pierwszym przebiegu tworzy się kod wynikowy „na próbę", nie wstawiając do niego stałych opisanych przez etykiety (nie są one bowiem jeszcze znane). W tym przebiegu ustala się ich wartości. W drugim przebiegu generuje się ostatecznie kod wynikowy, ponieważ wartości etykiet są już ustalone. Tablica 4.6 na stronie 108 zawiera program w asemblerze obliczający największy wspólny podzielnik dwóch liczb według algorytmu z tablicy 4.4, tablica 4.7 jego tłumaczenie na kod wynikowy, a tablica 4.5 na sąsiedniej stronie plik z kodem wynikowym w postaci szesnastkowej.

4.3. Zadania

Zadanie 4.1. Zaprogramuj poniższe problemy w kodzie maszynowym i w asemblerze procesora Sextium II:

1. Program wczytujący liczbę całkowitą i wypisujący liczbę jej nietrywialnych (różnych od 1 i od niej samej) podzielników.
2. Program wczytujący trzy liczby x, y i z i wyliczający $x^y \bmod z$.
3. Program wczytujący dwie liczby x i y i wyliczający $x! \bmod y$.
4. Program wczytujący liczby aż do przeczytania liczby 0 i wypisujący sumę wczytanych liczb.
5. Program wczytujący sześć liczb całkowitych opisujących dwie chwile w ciągu doby w formacie godzina-minuta-sekunda i wypisujący liczbę sekund dzielących obie chwile (ujemną, jeśli druga chwila jest wcześniejsza niż pierwsza).
6. Program wczytujący liczbę całkowitą będącą numerem roku i wypisujący liczbę dni tego roku, tj. 365 lub, dla lat przestępnych, 366 (rok jest przestępny, jeśli jego numer dzieli się przez 4 i nie dzieli się przez 100 lub dzieli się przez 400).
7. Program wczytujący sześć liczb całkowitych opisujących dwie daty w formacie dzień-miesiąc-rok i wypisujący liczbę dni dzielących te daty (ujemną, jeśli druga data jest wcześniejsza niż pierwsza).

```
A532 001C A532 001D 9516 001D A568 0019
6A51 001C ED6A 001C 51C6 A562 001E A516
001D A562 001C A516 001E A562 001D A700
0004 001C F000 0000 0000 0000
```

Tablica 4.5. Plik `gcd.hex` zawierający program dla procesora Sextium II w postaci szesnastkowej obliczający największy wspólny podzielnik dwóch liczb (opis w tablicy 4.7 na stronie 109)

Zadanie 4.2. Napisz *emulator*[3] procesora Sextium II, tj. program, który z podanego w wierszu poleceń pliku binarnego wczytuje program dla procesora Sextium II i symuluje jego pracę. Pliki binarne w komputerach mają zwykle strukturę bajtową, przyjmujemy więc, że słowa zajmują po dwa kolejne bajty. Rozkazy wejścia (wyjścia) powinny powodować wczytywanie (wypisywanie) liczb całkowitych w zapisie dziesiętnym ze znakiem, po jednej w wierszu, z (do) standardowego strumienia wejściowego (wyjściowego). Dla ułatwienia wpisywania programów binarnych napisz programy `hex2bin` i `bin2hex`, które dokonują konwersji pomiędzy zapisem szesnastkowym i binarną wersją pliku. Program `hex2bin` pomija białe znaki (spacje, znaki tabulacji i nowego wiersza) i przyjmuje, że dwie kolejne cyfry szesnastkowe opisują jeden bajt. Program `bin2hex` w każdym wierszu wypisuje zawartość 8 słów, tj. 32 cyfry szesnastkowe w grupach po 4 cyfry oddzielone spacją. Program wczytujący dwie liczby z wejścia i wypisujący ich sumę zapiszemy w postaci szesnastkowej jako:

```
363B 4F00
```

a program obliczający największy wspólny podzielnik dwóch liczb w postaci szesnastkowej jest przedstawiony w tablicy 4.5. Na podstawie takiego pliku program `hex2bin` utworzy czterobajtowy plik wykonywalny. Przyjmijmy, że domyślnie nazwa programu w zapisie szesnastkowym powinna mieć rozszerzenie `*.hex`, program binarny zaś rozszerzenie `*.sextium`.

Zadanie 4.3. Napisz asembler dla procesora Sextium II, tj. program przetwarzający kod źródłowy (domyślnie zapisany w pliku z rozszerzeniem `*.asm`) w asemblerze na postać binarną. Napisz następnie *deasembler*, tj. program, który dokonuje sztuczki (prawie) odwrotnej, tj. dla zadanej postaci binarnej wypisuje tekst przypominający asembler (tylko zamiast oryginalnych etykiet w programie występują rzeczywiste adresy instrukcji zapisane w postaci czterech cyfr szesnastkowych).

[3] Emulator to program symulujący działanie procesora A napisany dla procesora B. Dzięki niemu można programy napisane dla procesora A uruchamiać na maszynie B. W praktyce używa się np. programu Wabi, dzięki któremu można uruchamiać na maszynach SPARC aplikacje napisane dla systemu Microsoft Windows działającego na komputerach PC. Jest również podobny program pozwalający na uruchamianie takich aplikacji na komputerach Macintosh. Miłośnicy zabawkowych mikrokomputerów ZX Spectrum, Atari itp. mogą dzięki emulatorom „wskrzeszać" swoje ulubione programy na współczesnych maszynach, mimo że ich stare Spektrusie i Atarusie już dawno dołączyły do grona zbawionych komputerów w Niebie.

```
# Program gcd.asm
# Wczytuje dwie liczby i wypisuje
# ich największy wspólny dzielnik
#
        CONST x         # czytaj x
        SWAPA
        READ
        STORE
        CONST y         # czytaj y
        SWAPA
        READ
        STORE
dalej:  CONST y         # czy y=0?
        SWAPA
        LOAD
        SWAPD
        CONST koniec
        SWAPA
        SWAPD
        BRANCHZ
        SWAPD           # z=x/y
        CONST x
        SWAPA
        LOAD
        DIV
        MUL             # z=z*y
        SWAPD           # z=x-z
        CONST x
        SWAPA
        LOAD
        SUB
        SWAPD
        CONST z
        SWAPA
        SWAPD
        STORE
        CONST y         # x=y
        SWAPA
        LOAD
        SWAPD
        CONST x
        SWAPA
        SWAPD
        STORE
        CONST z         # y=z
        SWAPA
        LOAD
        SWAPD
        CONST y
        SWAPA
        SWAPD
        STORE
        CONST dalej
        JUMP
koniec: CONST x         # pisz x
        SWAPA
        LOAD
        WRITE
        HALT
     x: DATA            # zmienne
     y: DATA
     z: DATA
```

Tablica 4.6. Program w asemblerze obliczający największy wspólny podzielnik dwóch liczb

	adres	zawartość słowa	opis zawartości słowa
	0000	A532	CONST SWAPA READ STORE
	0001	001C	adres reprezentowany przez zmienną x
	0002	A532	CONST SWAPA READ STORE
	0003	001D	adres reprezentowany przez zmienną y
dalej =	0004	9516	CONST SWAPA LOAD SWAPD
	0005	001D	adres reprezentowany przez zmienną y
	0006	A568	CONST SWAPA SWAPD BRANCHZ
	0007	0019	adres reprezentowany przez zmienną koniec
	0008	6A51	SWAPD CONST SWAPA LOAD
	0009	001C	adres reprezentowany przez zmienną x
	000A	ED6A	DIV MUL SWAPD CONST
	000B	001C	adres reprezentowany przez zmienną x
	000C	51C6	SWAPA LOAD SUB SWAPD
	000D	A562	CONST SWAPA SWAPD STORE
	000E	001E	adres reprezentowany przez zmienną z
	000F	A516	CONST SWAPA LOAD SWAPD
	0010	001D	adres reprezentowany przez zmienną y
	0011	A562	CONST SWAPA SWAPD STORE
	0012	001C	adres reprezentowany przez zmienną x
	0013	A516	CONST SWAPA LOAD SWAPD
	0014	001E	adres reprezentowany przez zmienną z
	0015	A562	CONST SWAPA SWAPD STORE
	0016	001D	adres reprezentowany przez zmienną y
	0017	A700	CONST JUMP NOP NOP
	0018	0004	adres reprezentowany przez zmienną dalej
koniec =	0019	A514	CONST SWAPA LOAD WRITE
	001A	001C	adres reprezentowany przez zmienną x
	001B	F000	HALT NOP NOP NOP
x =	001C	0000	miejsce przechowywania zmiennej x
y =	001D	0000	miejsce przechowywania zmiennej y
z =	001E	0000	miejsce przechowywania zmiennej z

Tablica 4.7. Tłumaczenie programu w asemblerze z tablicy 4.6 na kod procesora Sextium II

Rozdział 5

Programowanie imperatywne

Do uzupełnienia:

5.1. Komórki pamięci i instrukcja przypisania
5.1.1. Skutki uboczne
5.1.2. L i R-wartości, wyłuskanie (niejawna dereferencja) na przykładzie języka Pascal
5.1.3. Wskaźniki i operator przypisania w języku Standard ML
5.1.4. Wskaźniki a adresy na przykładzie Pascala i C
5.2. Typy danych
5.2.1. Typy proste
5.2.2. Koercje
5.2.3. Agregaty: tablice, struktury i unie. Przydział pamięci dla nich
5.3. Automatyczne zarządzanie pamięcią i odśmiecanie
5.4. Operacje wejścia/wyjścia
5.4.1. Operacje wejścia/wyjścia w języku Standard ML

Literatura zastępcza: [101]

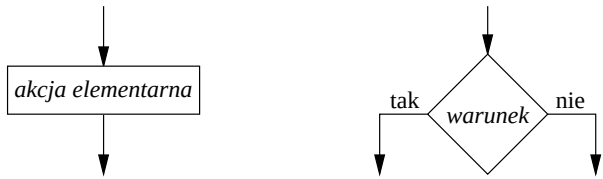
Rozdział 6

Programowanie strukturalne

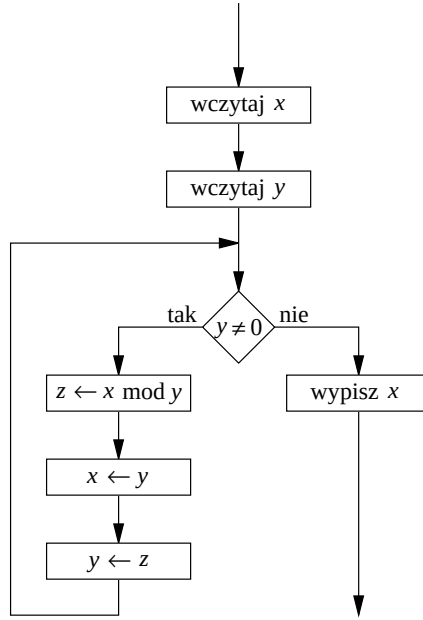
Jedynym ratunkiem jest struktura.
Niklaus Wirth, *Modula 2*

6.1. Schematy blokowe programów

W opisach algorytmów można wydzielić czynności dwóch zasadniczych rodzajów: pojedyncze, elementarne akcje, których zestaw jest zależny od tego, jakie czynności uznamy za niepodzielne oraz podejmowanie dycyzji o kolejności wykonywania takich akcji w zależności od spełnienia pewnych warunków. W ten sposób można podzielić rozkazy języka maszynowego (np. procesora Sextium opisanego w rozdziale 4). Do zbioru rozkazów zmieniających kolejność wykonywania czynności zaliczymy rozkazy skoków, zaś do zbioru akcji elementarnych — pozostałe rozkazy. Algorytmy przedstawiane w notacji „wysokiego poziomu", np. algorytm Euklidesa obliczania największego wspólnego podzielnika z tablicy 4.4 na stronie 106 również można opisać w podobny sposób. W tym przypadku za akcje elementarne jesteśmy gotowi uznać całe instrukcje przypisania (mimo, że ich wykonanie może pociągać obliczenie skomplikowanego wyrażenia arytmetycznego), wejścia/wyjścia itp., a pojedyncze decyzje w algorytmie mogą być podejmowane na podstawie wartości logicznej dowolnie skomplikowanych wyrażeń logicznych. Wybór zestawu akcji elementarnych i postaci warun-



Rysunek 6.1. Elementy schematów blokowych programów



Rysunek 6.2. Schemat blokowy algorytmu obliczania największego wspólnego podzielnika z tablicy 4.4 na stronie 106

ków, które można testować zależy od poziomu szczegółowości, z jakim chcemy analizować nasze algorytmy.

O ile analiza programu złożonego z ciągu następujących po sobie akcji elementarnych nie przedstawia trudności, to czytanie programu zawierającego skoki do odległych miejsc programu może być żmudne. Dlatego algorytmy wygodnie jest przedstawiać w postaci tzw. *schematów blokowych* (*diagramów przepływu*, ang. *flow diagrams*), w których akcje elementarne są reprezentowane przy pomocy prostokątów a sprawdzanie warunków przy pomocy rombów (zob. rysunek 6.1 na poprzedniej stronie). Te prostokąty i romby łączy się strzałkami tak, że postępując wzdłuż strzałek można odtworzyć kolejność wykonania akcji elementarnych. Dla przykładu schemat blokowy algorytmu Euklidesa z tablicy 4.4 jest przedstawiony na rysunku 6.2.

6.2. Instrukcje strukturalne

Przyjęcie, że akcją elementarną jest obliczenie dowolnego (być może skomplikowanego) wyrażenia wydaje się być założeniem gwarantującym właściwy stopień szczegółowości, gdy analizujemy programy napisane w językach tzw. „wysokiego poziomu”. Co prawda oblicze-

nie wyrażenia wymaga zwykle wykonania wielu rozkazów maszyny, jednak algorytm obliczania wyrażeń (przeważnie z użyciem stosu) jest stosunkowo prosty (por. podrozdział 3.4) i nie musimy za każdym razem analizować sposobu jego działania.

Historycznie pierwszy język „wysokiego poziomu", FORTRAN (ang. *FORmula TRANslator*, czyli translator wyrażeń), pozwalał na zapis dowolnie skomplikowanych wyrażeń w notacji zbliżonej do notacji matematycznej, a kompilator sam tłumaczył te wyrażenia na ciągi rozkazów procesora. Natomiast instrukcje zmieniające kolejność wykonania czynności, tj. tzw. *instrukcje sterujące*, były zbliżone do rozkazów maszynowych. FORTRAN był więc po asemblerze reprezentantem kolejnego szczebla w hierarchii „poziomu" języków programowania — uwalniał programistę od konieczności żmudnego kodowania obliczeń za pomocą rozkazów maszyny, jednak przepływ sterowania w programie nadal był kontrolowany na poziomie języka maszynowego. Jeśli potraktujemy instrukcję przypisania w FORTRAN-ie jako akcję elementarną i będziemy używać logicznej instrukcji warunkowej (składnia FORTRAN-u 66)

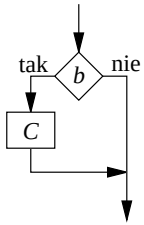
IF (*warunek*) GOTO *etykieta*

do zmiany kolejności wykonania instrukcji, to otrzymamy idealną odpowiedniość między programami w FORTRAN-ie i ich schematami blokowymi (nb. schematy blokowe zostały wymyślone w latach 50-tych zeszłego wieku i stały się wśród programistów popularnym narzędziem służącym właśnie do czytelnego przedstawiania programów zapisanych z użyciem skoków).

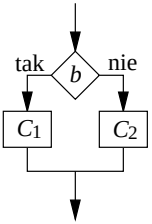
Gdy rozważamy programy w języku C (zob. [82, 83]), również wygodnie jest przyjąć obliczenie wyrażenia jako akcję elementarną. W szczególności za akcję elementarną możemy uważać *instrukcję wyrażeniową* (którą jest najczęściej wyrażenie przypisania zakończone średnikiem). Do jawnej zmiany kolejności wykonania instrukcji w języku C służą *instrukcje skoku*: `goto`, `break`, `continue` i `return`. Pierwsza jest instrukcją bezwarunkowego przekazania sterowania do miejsca opatrzonego etykietą, dwie następne są związane z obsługą pętli. O instrukcji `return` mówi się w kontekście przekazywania sterowania w funkcjach, nie będziemy się nią zatem w tym miejscu zajmować. Zestaw instrukcji sterujących języka C nie kończy się bynajmniej na instrukcjach skoku, mamy bowiem instrukcję *złożoną*, instrukcje *wyboru* (`if`, `if-else` i `switch`) i instrukcje *powtarzania* (`while`, `do` i `for`). Instrukcje te mają swoją wewnętrzną strukturę, rozpadają się na wiele akcji elementarnych. Odpowiadające im schematy blokowe są przedstawione na rysunku 6.3 na następnej stronie. Można zadać pytanie, dlaczego w języku C wprowadzono aż tyle instrukcji sterujących, skoro sama instrukcja `goto` skojarzona z instrukcją `if` pozwala na realizowanie skoków warunkowych, a zatem swobodne przekazywanie sterowania w programie w zależności od spełnienia podanych warunków. Instrukcję postaci

`if (`*b*`) goto `*l*`;`

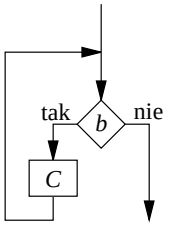
można ponadto w łatwy sposób przetłumaczyć na kod maszynowy procesora, posiada on bowiem rozkazy o zbliżonym znaczeniu. Taka pojedyncza instrukcja skoku warunkowego wraz z instrukcjami przypisania jako akcjami elementarnymi pozwala na zapisanie w języku



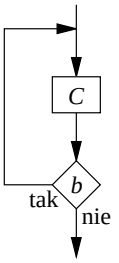
`if (`b`)` C



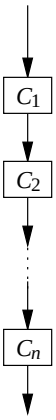
`if (`b`)` C_1 `else` C_2



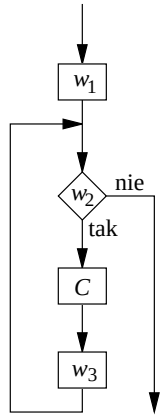
`while (`b`)` C



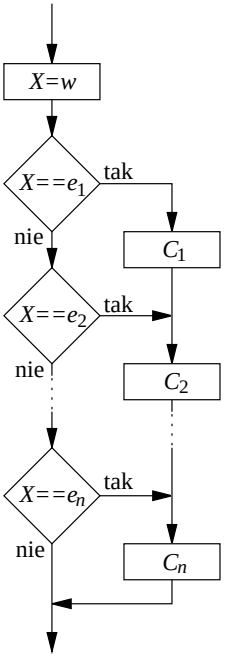
`do` C `while (`b`)`



`{` C_1 `;` C_2 `;` ... C_n `;` `}`



`for (`w_1`;`w_2`;`w_3`)` C



`switch (`w`) {`
`case` e_1`:`C_1`;`
`case` e_2`:`C_2`;`
⋮
`case` e_n`:`C_n`;`
`}`

Rysunek 6.3. Schematy blokowe instrukcji strukturalnych języka C

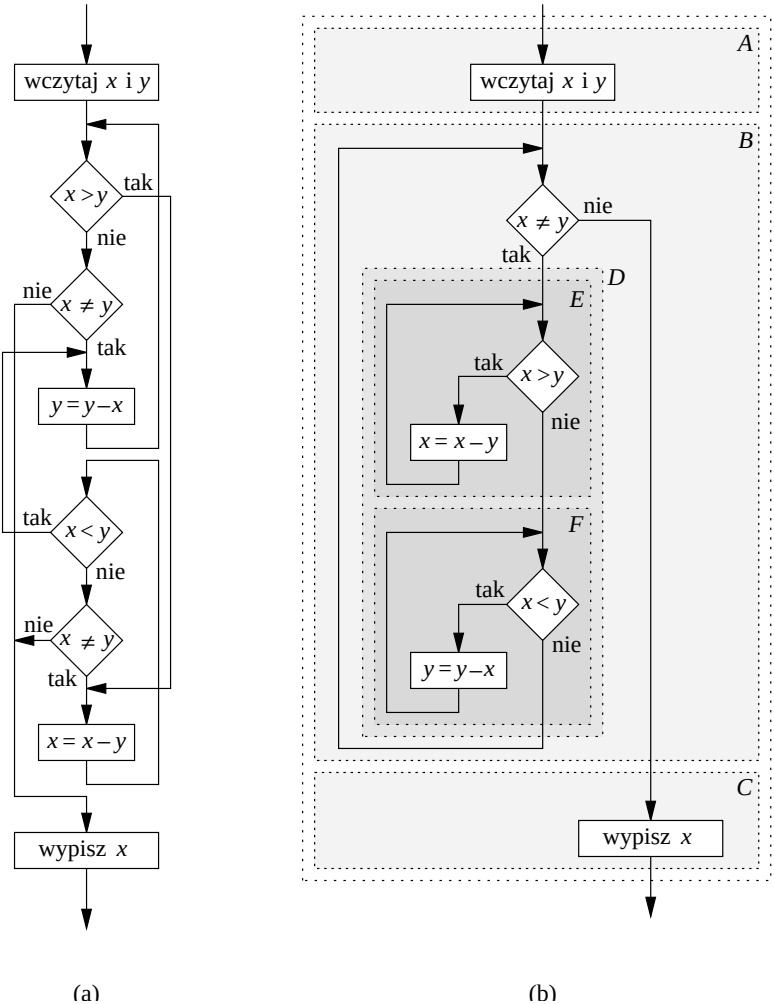
programowania dowolnego algorytmu. Jednak już w latach 60-tych zeszłego wieku zauważono, że programy napisane z użyciem dużej liczby skoków są nieczytelne, a ich analiza jest bardzo trudna [43]. Rozważmy dla przykładu kolejną wersję programu obliczającego największy wspólny podzielnik dwóch liczb dodatnich, którego schemat blokowy jest przedstawiony na rysunku 6.4 (a):

```
#include<stdio.h>
int main ()
{
   int x, y;

               scanf ("%d%d", &x, &y);
   maly:       if (x>y) goto zmniejszx;
               if (x==y) goto koniec;
   zmniejszy: y=y-x;
               goto maly;
   duzy:       if (x<y) goto zmniejszy;
               if (x==y) goto koniec;
   zmniejszx: x=x-y;
               goto duzy;
   koniec:     printf ("%d\n", x);
               return 0;
}
```

Ani powyższy program, ani jego schemat blokowy nie są łatwe do zrozumienia.[1] Na rysunku 6.4 (b) jest przedstawiony inny, dużo czytelniejszy schemat blokowy. Oba schematy z rysunku 6.4 są przy tym równoważne w tym sensie, że dla każdych danych wejściowych ciąg wykonanych akcji elementarnych będzie taki sam (jednak liczba i kolejność sprawdzania warunków może być inna). Nie chcemy tu wdawać się w dyskusję na temat ścisłej definicji równoważności schematów. Intuicyjnie oba schematy przedstawiają ten sam algorytm. Schemat z rysunku (b) ma ciekawą własność: ma prostą wewnętrzną *strukturę*. Na najwyższym poziomie składa się z sekwencji trzech mniejszych schematów: *A*, *B* i *C*. Schematy *A* i *C* zawierają pojedyncze akcje elementarne. Z kolei schemat *B* ma strukturę schematu pętli `while` z rysunku 6.3, w którym akcją do wykonania jest schemat *D*. Ten znowuż jest sekwencją schematów *E* i *F*, a one same znów mają strukturę pętli `while`.

Zauważmy, że wszystkie schematy z rysunku 6.3 mają dokładnie jedno wejście i dokładnie jedno wyjście. Składanie ich ze sobą zachowuje tę własność. Jeżeli program jest zbudowany wyłącznie ze schematów o jednym wejściu i jednym wyjściu, wówczas nie musimy go analizować w całości. Jeżeli bowiem rozważymy jego fragment i ustalimy, jakie jest jego znaczenie, skoro fragment ten ma tylko jedno wejście i jedno wyjście, możemy go od tej chwili traktować jako nową, niepodzielną akcję elementarną „wyższego poziomu", pewną „czarną skrzynkę", do której wnętrza nie musimy już zaglądać. Dzięki temu potrafimy utrzymać stopień komplikacji schematu, jaki w danej chwili analizujemy, na poziomie

[1] Autor przyznaje, że dołożył starań, by program był nieczytelny.



Rysunek 6.4. Schematy blokowe dwóch programów obliczających największy wspólny podzielnik

dostatecznie niskim, byśmy mogli nad nim intelektualnie zapanować.[2] W tym sensie schemat z rysunku 6.4 (b) składa się jedynie z sekwencji trzech akcji: *A*, *B* i *C*. Teraz możemy skupić uwagę *osobno* na opisie znaczenia każdej z trzech wymienionych akcji. Schemat *A* zawiera akcję elementarną. Schemat *B* zaś ma strukturę pętli `while`, w której warunkiem jest $x \neq y$, zaś akcją jest schemat *D* itd. Takiej hierarchicznej analizy nie jesteśmy w stanie wykonać dla schematu z rysunku 6.4 (a) dlatego, że dowolnie „rozrzucone" miejsca docelowe skoków nie pozwalają nam skupić uwagi na drobnym fragmencie programu zapominając na jakiś czas o jego reszcie.

Idea konstruowania oprogramowania w sposób systematyczny została rozwinięta w latach 70-tych zeszłego wieku i stanowi podstawę współczesnej inżynierii oprogramowania (rzetelnej wiedzy o programistycznym rzemiośle). Ponieważ nasze rozważania skupiają się raczej na teorii języków programowania niż na inżynierii, dalsze wskazówki, jak tworzyć niezawodne oprogramowanie Czytelnik zechce znaleźć w literaturze [37, 189, 79, 123, 192, 142, 166, 34, 104, 81, 20, 45, 7, 178]. Poniżej podajemy jedynie podstawową terminologię.

Schemat strukturalny: schemat blokowy o jednym wejściu i jednym wyjściu, którego komponenty też mają tę własność. Dzięki temu ma on prostą, hierarchiczną *strukturę*.

Instrukcja strukturalna: instrukcja sterująca, której odpowiada strukturalny schemat blokowy.

Projektowanie strukturalne: projektowanie programu z użyciem struktur sterujących o jednym wejściu i jednym wyjściu. Dokonuje się przy tym dekompozycji problemu na mniejsze, analizowane niezależnie fragmenty, co pozwala zmniejszyć stopień komplikacji programu i nadać mu czytelną, hierarchiczną *strukturę*. Używa się przy tym albo metody *wstępującej*, albo *zstępującej*.

Metoda wstępująca: metoda projektowania strukturalnego, w której budujemy najpierw małe komponenty programu, a następnie z nich coraz większe fragmenty, tak długo, aż w końcu otrzymamy cały program.

Metoda zstępująca: metoda projektowania strukturalnego, w której rozpoczynamy od naszkicowania struktury całego programu, a następnie precyzujemy coraz drobniejsze detale tak długo, aż w końcu dojdziemy do podproblemów, które potrafimy rozwiązać w jednym kroku. Przykład zstępującej metody budowania algorytmu znajdzie Czytelnik w podrozdziale 3.6.5, w którym użyto tej metody do zbudowania algorytmu dokonującego przekładu wyrażeń z notacji infiksowej na postfiksową.

Kodowanie strukturalne: zapis programu z użyciem instrukcji strukturalnych. Używa się przy tym specjalnych konwencji zapisu programów tak, by układ tekstu optycznie ujawniał strukturę programu (np. stosuje się *wcięcia*).

Wcięcia: konwencja zapisu programów, polegająca na przesuwaniu lewego marginesu w programie w prawo o kilka spacji (zwykle dwie lub trzy) za każdym razem, gdy zwiększa się poziom zagnieżdżenia struktur sterujących.

[2] Psychologowie mówią o magicznej, nieprzekraczalnej liczbie *siedmiu* obiektów, na których można *jednocześnie* skupić uwagę [113].

Konstruowanie oprogramowania metodą systematyczną: konstruowanie programu z wykorzystaniem metod projektowania i kodowania strukturalnego.

Strukturalny język programowania: język programowania, który posiada strukturalne instrukcje sterujące i przez to swoją budową wspiera strukturalny styl programowania. Trzeba jednak podkreślić, że strukturalnie *można* programować także w języku, w którym jedyną instrukcją sterującą jest skok warunkowy i odwrotnie, można pisać niestrukturalne programy w języku strukturalnym.

Wśród struktur sterujących o jednym wejściu i jednym wyjściu wyróżnia się zwykle zestaw kilku najprostszych, które są odpowiednikami złożenia sekwencyjnego instrukcji, instrukcji warunkowej z frazą `else` i instrukcji `while` (por. rysunek 6.3 na stronie 116). Nazywa się je niekiedy D-strukturami (na cześć Edsgera Dijkstry). Formalnie D-struktury definiuje się indukcyjnie:

1. akcje elementarne są D-strukturami;
2. złożenie sekwencyjne D-struktur jest D-strukturą;
3. struktura warunkowa, w której obu gałęziach znajdują się D-struktury jest D-strukturą;
4. struktura pętli `while`, która we wnętrzu pętli zawiera D-strukturę jest D-strukturą;
5. wszystkie D-struktury można zbudować korzystając z reguł opisanych w punktach 1–4.

Niekiedy do powyższego zbioru dodaje się inne, bardziej złożone struktury, np. wszystkie pozostałe struktury przedstawione na rysunku 6.3 na stronie 116 (z wyjątkiem instrukcji `switch`, która nie jest w pełni strukturalna — zamiast niej można dodać instrukcję **case** z Pascala). Takie struktury bywają nazywane D'-strukturami. Dowolne schematy blokowe są nazywane L-strukturami.

Aby nie tylko wspierać, ale wręcz wymusić strukturalny styl programowania, struktury sterujące w niektórych językach (np. w Moduli 2) są wyłącznie D'-strukturami. W innych pozostawiono instrukcję skoku `goto`, wstydliwie się ją jednak ukrywa. Narzuca się zatem pytanie, czy D-struktury (lub D'-struktury) wystarczają do zapisu każdego algorytmu. W szczególności można zapytać, czy każdą L-strukturę można przekształcić do równoważnej[3] D-struktury.

Rozważmy język J_L, w którym programy są zbudowane z pewnych akcji elementarnych zawierających co najmniej instrukcje przypisania postaci `X = e`, przy czym wyrażenia *e* mogą być przynajmniej postaci `X+1` lub *i*, gdzie `X` jest zmienną, zaś *i* stałą i instrukcji skoku warunkowego

`if` *b* `goto` *l*

gdzie *b* jest wyrażeniem logicznym co najmniej postaci `X==`*i* i `X<=`*i*, *l* zaś — etykietą. Bez zmniejszania ogólności rozważań możemy przyjąć, że każda instrukcja ma dokładnie jedną etykietę i że etykiety te są kolejnymi liczbami naturalnymi (etykieta jest numerem instrukcji w programie). W języku J_L możemy zapisać dowolną L-strukturę, nie nałożyliśmy bowiem

[3] Dalej nie zdefiniowaliśmy, co to znaczy!

```
N = 1;
while (N<=k) {
    if (N==1) I'_1
        ⋮
    if (N==i) I'_i
        ⋮
    if (N==k) I'_k
}
```

Rysunek 6.5. Schemat transformacji programu do D-struktury

żadnych ograniczeń na miejsca docelowe skoków. Rozważmy teraz język programowania J_D, w którym instrukcjami sterującymi są instrukcja wyboru `if` (bez frazy `else`) i instrukcja `while`, przy czym akcje elementarne są w obu językach takie same, a w instrukcjach `if` i `while` można testować takie same warunki. Rozważmy program w języku J_L składający się z $k \geq 1$ instrukcji. Niech `N` będzie identyfikatorem nie występującym w tym programie. Przekształcimy dany program do równoważnego mu (w sensie, który za chwilę stanie się jasny) programu w języku J_D. Program w w tym języku będzie miał strukturę przedstawioną na rysunku 6.5, gdzie instrukcje I'_i dla $i = 1, \ldots, k$ powstają z instrukcji I_i programu w języku J_L w następujący sposób: jeżeli i-ta instrukcja I_i programu w języku J_L jest akcją elementarną A_i, to instrukcja I'_i jest postaci:

$$\{A_i\texttt{; N = N+1;}\}$$

Jeżeli zaś instrukcja I_i jest instrukcją skoku warunkowego

$$\texttt{if (}b\texttt{) goto }l\texttt{;}$$

to odpowiada jej instrukcja

$$\{\texttt{N = N+1; if (}b\texttt{) N = }l\texttt{;}\}$$

(przypomnijmy, że etykieta l jest liczbą z przedziału $1, \ldots, k$). W nowym programie zmienna `N` przechowuje numer kolejnej instrukcji do wykonania. Zauważmy, że tak otrzymany program w języku J_D jest równoważny z wyjściowym programem w języku J_L w tym sensie, że dla każdych danych wejściowych kolejność wykonywania akcji elementarnych i kolejność sprawdzania warunków w obu programach będzie taka sama, z tym, że w nowym programie (a) będzie używana dodatkowa zmienna `N`, (b) będą sprawdzane dodatkowe warunki postaci `N==`i i `N<=`k, oraz (c) będą wykonywane dodatkowe akcje elementarne postaci `N=1`, `N=N+1` i `N=`l.

6.3. Strukturalne instrukcje skoku

Podaliśmy teoretyczne uzasadnienie faktu, że D-struktury są wystarczające do zapisu dowolnych algorytmów. Co więcej: każdy algorytm można zapisać używając instrukcji pętli `while` tylko raz. Ponadto stopień zagnieżdżenia struktur sterujących może wynosić nie więcej niż 7. Wynik ten, opublikowany w latach 60-tych przez Böhma i Jacopiniego [24] ma się jednak nijak do programistycznej praktyki: otrzymany program jest co prawda zbudowany z użyciem wyłacznie strukturalnych instrukcji sterujących, jednak w żadnym razie nie jest strukturalnie zbudowany!

Rozważmy fragment programu w języku Pascal, który zamienia ciąg cyfr na liczbę całkowitą i wypisuje resztę z dzielenia jej przez 13. Ciąg cyfr jest wczytywany z pliku f aż do napotkania znaku nie będącego cyfrą lub końca pliku. Pusty ciąg cyfr jest interpretowany jako liczba 0. Ponieważ w Pascalu (inaczej niż w języku C) nie wolno próbować czytać znaku z pliku, który został przewinięty do końca, przed każdym wykonaniem operacji *read* należy sprawdzić, wywołując funkcję *eof*, czy wskaźnik pliku nie znajduje się już na jego końcu. To sugeruje, by zorganizować obliczenia w postaci pętli **while** z warunkiem **not** *eof*(f). Po wczytaniu kolejnego znaku należy sprawdzić, czy jest on cyfrą i jeśli nie, to przerwać wykonanie pętli. Program zbudowany w ten sposób jest przedstawiony w tablicy 6.1. Ponieważ możemy opuścić ciało pętli po wykonaniu procedury *read* ale przed obliczeniem nowej wartości *n*, ciało pętli jest strukturą o dwóch wyjściach. Nie jest więc D-strukturą. Spróbujmy zatem zbudować D-strukturę wykonującą te same obliczenia. Okazuje się, że nie uda nam się tego zrobić bez wprowadzenia dodatkowych czynności i/lub zmiennych. Jedno z możliwych rozwiązań, przedstawione w tablicy 6.2, polega na wprowadzeniu pomocniczej zmienej logicznej *dalej*, która na początku ma wartość *true* i jest dodatkowym warunkiem wejścia do pętli. Zostaje jej przypisana wartość *false*, gdy kolejny przeczytany znak nie jest cyfrą.

Można mieć wątpliwości, czy strukturalny program z tablicy 6.2 jest bardziej czytelny od zawierającego skok programu z tablicy 6.1. Schemat programu o dwóch wyjściach, w którym następuje skok z wnętrza pętli do instrukcji następującej bezpośrednio po niej pojawia się w programowaniu na tyle często, że w wielu językach programowania (np. w C) wprowadzono specjalną instrukcję skoku `break` realizującą ten schemat. Obok instrukcji `break` w wielu językach występuje też instrukcja `continue`, która przenosi sterowanie na koniec pętli tak, że następnym krokiem jest sprawdzenie warunku wejścia do pętli (wykonanie pętli jest kontynuowane). Pozwala ona uniknąć zagnieżdżania wielu instrukcji `if`, jednak jej znaczenie jest mniejsze niż istrukcji `break`. Schematy instrukcji `break` i `continue` w języku C są przedstawione na rysunku 6.6 na stronie 124.

Pisząc program nie należy zapominać, że instrukcje `break` i `continue` są w istocie instrukcjami skoku i zaburzają proces strukturalnej dekompozycji programu na podstruktury (ponieważ mają więcej niż jedno wyjście), jednak stosowane z umiarem przyczyniają się do uproszczenia i skrócenia programów, a przecież o to właśnie chodzi.

W niektórych językach, np. w Adzie, istnieje możliwość opuszczenia wielu zagnieżdżonych pętli. W Adzie do tego celu służy instrukcja **exit**. Instrukcja pętli **loop** w Adzie może mieć etykietę (identyfikator oddzielony dwukropkiem od słowa **loop**). Wykonanie instrukcji **exit** l znajdującej się w zasięgu kilku zagnieżdżonych pętli powoduje przerwanie wykonania

```
        n := 0;
        while not eof(f) do
        begin
            read(f, c);
            if c < '0' or c > '9'
                then goto koniec;
            n := 10 × n + chr(ord(c) − ord('0'))
        end;
koniec: writeln(n mod 13);
```

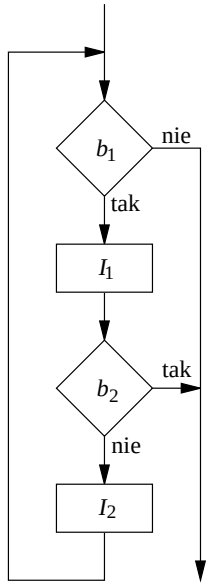
Tablica 6.1. Fragment programu w języku Pascal nie będący D-strukturą

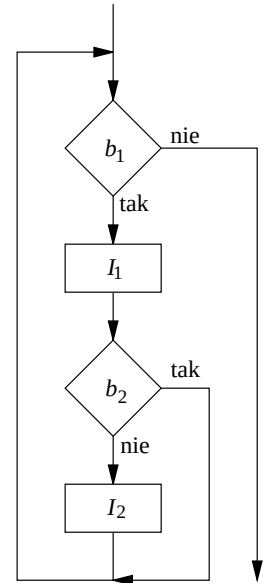
```
dalej := true;
n := 0;
while dalej and not eof(f) do
begin
    read(f, c);
    if c < '0' or c > '9'
        then dalej := false;
        else n := 10 × n + chr(ord(c) − ord('0'));
end;
writeln(n mod 13);
```

Tablica 6.2. Jedna z metod przekształcenia programu z tablicy 6.1 do D-struktury przez wprowadzenie dodatkowej zmiennej logicznej

```
N := 0;
while not TEXT_IO.END_OF_FILE (F)
loop
    TEXT_IO.GET (F, C);
    exit when C < '0' or else C > '9';
    N := 10 * N + CHARACTER'VAL (CHARACTER'POS (C) −
            CHARACTER'POS ('0'));
end loop;
INTEGER_IO.PUT (N mod 13);
```

Tablica 6.3. Fragment programu z tablicy 6.1 w języku Ada, w którym zamiast instrukcji skoku **goto** użyto strukturalnej instrukcji skoku **exit**



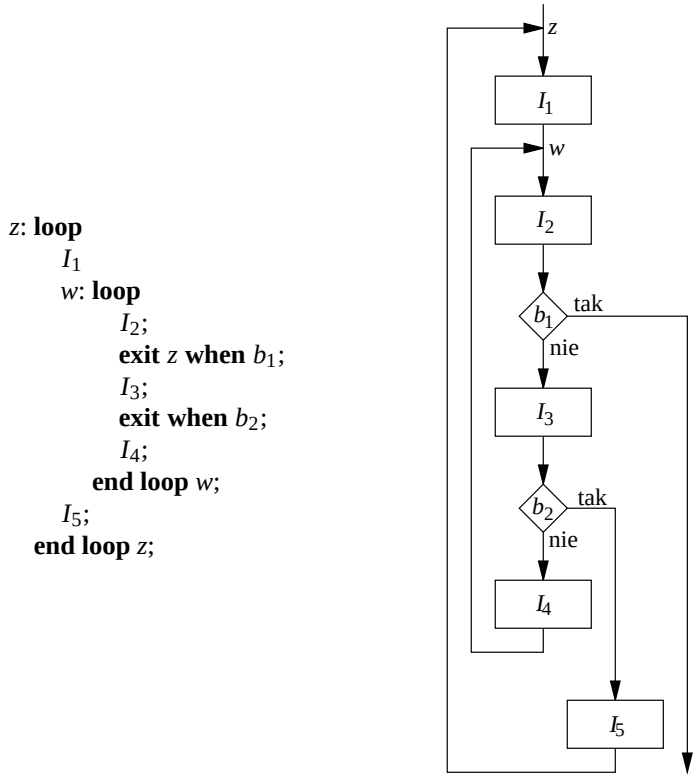


`while (`b_1`) {`I_1`; if (`b_2`) break;` I_2`;}` `while (`b_1`) {`I_1`; if (`b_2`) continue;` I_2`;}`

Rysunek 6.6. Instrukcje skoku `break` i `continue` w języku C

pętli o etykiecie *l*. Instrukcja **exit** bez etykiety ma podobne znaczenie, jak instrukcja `break` w języku C — przerywa działanie najbardziej wewnętrznej pętli obejmującej ją. Instrukcja **exit** może być zakończona opcjonalną frazą **when**, po której następuje warunek logiczny. Instrukcja **exit** *l* **when** *b* jest równoważna instrukcji **if** *b* **then exit** *l*. Ponieważ z instrukcji **exit** korzysta się prawie zawsze warunkowo, wprowadzono tę dodatkową, bardziej czytelną składnię. Wersja programu z tablicy 6.1 w Adzie z użyciem instrukcji skoku **exit** jest przedstawiona w tablicy 6.3, zaś przykładowy schemat programu zawierającego instrukcje **exit** w Adzie jest przedstawiony na rysunku 6.7. Jeżeli warunek b_1 będzie spełniony, instrukcja **exit** spowoduje zakończenie wykonania obu pętli. Jeżeli spełniony będzie warunek b_2, zostanie zakończone wykonanie wewnętrznej pętli *w*. Zamiast instrukcji **exit when** b_2 można też napisać **exit** *w* **when** b_2. Podobne instrukcje `break` *l* i `continue` *l*, gdzie *l* jest etykietą pętli występują w Javie.

Opisany wcześniej teoretyczny wynik Böhma-Jacopiniego dowodzi, że D-struktury wystarczą do zaprogramowania rozwiązania każdego problemu, który jest obliczalny (i faktycznie udało nam się znaleźć równoważną D-strukturę dla programu ze skokiem z tablicy 6.1). Interesuje nas jednak pytanie, czy D-struktury wystarczają do *czytelnego* zapisu wszelkich algorytmów. Aby rozważać ten problem ściśle, należałoby wpierw zdefiniować matematycznie pojęcie *czytelności*! Jedno z możliwych podejść polega na zadaniu pytania, czy każdą L-strukturę można przebudować do D-struktury jedynie przez zmianę następstwa akcji elementarnych, bez wprowadzania nowych akcji, warunków czy zmiennych. Okazuje się, że



Rysunek 6.7. Użycie instrukcji skoku **exit** w Adzie

odpowiedź na to pytanie jest negatywna [86, 105, 101]. D-struktury rozszerzone o instrukcję `break` bywają nazywane RE_1-strukturami, struktury umożliwiające zakończenie dowolnie wielu zagnieżdżonych pętli (tak jak w Adzie) RE_∞-strukturami. Wiadomo, że każdą D-strukturę można przekształcić do równoważnej RE_∞-struktury bez dodawania dodatkowych zmiennych lub akcji, jedynie przez zmianę kolejności instrukcji [86]. Jest to teoretyczne potwierdzenie przydatności instrukcji `break` w języku C i **exit** w Adzie. Wydaje się, że również w praktyce instrukcje `break` i **exit** wystarczają do czytelnego zapisu dowolnych programów.

6.4. Wyjątki

Programy są implementacjami algorytmów. Projektując algorytmy zwykle nie bierzemy pod uwagę sytuacji wyjątkowych związanych z awarią komputera (tj. przyjmujemy, że zarówno maszyna, jak i działające na niej programy są *niezawodne*) i zakładamy, że wszystkie akcje elementarne, z których budujemy te algorytmy będą wykonane pomyślnie. Na etapie projektowania algorytmu jest to bardzo rozsądne założenie. Jednak w praktyce wystąpienie

pewnych okoliczności uniemożliwia czasem kontynowanie wykonania algorytmu. Przykładem może być błąd operacji wejścia/wyjścia powstały na skutek uszkodzenia dyskietki itp. Sam program może również zawierać błedy przeoczone przez programistę, które mogą się ujawnić dopiero podczas jego biegu (indeks tablicy poza zakresem, dzielenie przez zero itd.) W takich sytuacjach wykonanie programu nie może być kontynuowane. W zamian powinny być podjęte odpowiednie akcje zastępcze. Ponieważ sytuacje awaryjne mogą powstać praktycznie w dowolnym miejscu, to program zawierający obsługę sytuacji nadzwyczajnych jest trudno zbudować w postaci strukturalnej. Dla przykładu w języku Turbo Pascal w wersji 3.0[4] programista może odpowiednimi *pragmami* (*dyrektywami kompilacji*) włączać i wyłączać kontrolę poprawności operacji wejścia/wyjścia. Jeżeli jest ona włączona, to każdy błąd powoduje natychmiastowe zatrzymanie programu. Można więc implementować algorytm nie przewidując żadnej obsługi sytuacji awaryjnych, licząc się jednak z tym, że program taki będzie zawodny. Jeżeli później zechcielibyśmy dopisać obsługę sytuacji awaryjnych, to możemy za pomocą odpowiedniej pragmy spowodować, by po wystąpieniu błędu wykonanie programu było kontynuowane, jednak wówczas po każdej operacji wejścia/wyjścia należy za pomocą instrukcji warunkowej sprawdzić, czy nie wystąpił błąd:

write(*f*, *s*);
if *IOResult* $\neq$ 0
    then kod obsługi błędu wejścia/wyjścia
    else dalsze instrukcje programu

Taki program jest usiany ogromną liczbą dozorów powyższej postaci, które zaciemniają jego strukturę. Skoro wykonanie normalnego ciągu akcji ma być przerwane, to „kod obsługi błędu wejścia/wyścia" jest prawdopodobnie skokiem do pewnego miejsca w programie, zawierającego podprogram obsługi takiego błędu. W większości języków programowania miejsce docelowe skoku musi występować w bieżąco wykonywanym bloku. Nie można zatem wyskoczyć na zewnątrz funkcji. W razie wystąpienia błędu sterowanie musi więc opuszczać zagnieżdżone wywołania funkcji po jedno po drugim. Aby ułatwić zaprogramowanie takiego wycofywania się z ciągu wywołań funkcji w języku C przyjęto zasadę (przykłady jej zastosowania znajdują się w podrozdziałach 3.4, 3.5, 3.6.3 i 3.6.5), że funkcje, w czasie wykonania których mogą powstać sytuacje awaryjne dostarczają w wyniku wartość logiczną zdania „podczas wykonania tej funkcji wystąpił błąd". Dla przykładu tak działa funkcja `fclose`. Inne funkcje w razie błędu dostarczają specjalną wartość, nie należącą do zbioru ich poprawnych wyników. Taką funkcją jest np. `fgetc`, która w razie błędu zamiast przeczytanego znaku zwraca w wyniku specjalną stałą `EOF`, różną od wszystkich znaków ASCII (dlatego typem wyniku funkcji `fgetc` jest `int` a nie `char`). Funkcja wczytująca dane z pliku może mieć w języku C postać

```
int wczytaj (FILE *f) {
   ...
   if ((ch=fgetc(f))==EOF)
      return 1;
```

[4] Późniejszych wersji nie znam, bo zacząłem programować w Standard ML-u.

```
   ...
   if (fclose(f))
      return 1;
   ...
   return 0;
}
```

Funkcja, która wywołuje funkcję `wczytaj`, również czyni to w instrukcji warunkowej:

```
   if (wczytaj(f))
      blad_we_wy();
```

Tutaj, podobnie jak w Turbo Pascalu, cały program jest usiany dozorami sprawdzającymi, czy nie wystąpił błąd. Często kod obsługi błędów dominuje swoim rozmiarem nad treścią samego algorytmu!

Opisane wyżej rozwiązania nie są satysfakcjonujące. Podstawowe cechy sytuacji awaryjnych są następujące:

1. taka sama sytuacja awaryjna (np. błąd zapisu do pliku) może powstać w wielu miejscach programu (np. w miejscu każdego wywołania funkcji zapisującej dane do pliku);
2. wykonanie ciągu instrukcji, w którym powstała taka sytuacja, musi być przerwane;
3. sterowanie powinno być przekazane do odpowiedniego podprogramu obsługi sytacji awaryjnej;
4. w różnych kontekstach podprogramy obsługi takiej samej sytuacji awaryjnej mogą być różne, nie sposób bowiem napisać np. uniwersalnego podprogramu obsługi błedu zapisu do pliku (bo zależy on od tego, co było zapisywane);
5. po zakończeniu obsługi sytuacji awaryjnej sterowanie *nie powraca* do przerwanego ciągu instrukcji (podprogram obsługi jest wykonywany *zamiast* dalszego ciągu instrukcji), przekazanie sterowania musi się zatem odbyć za pomocą skoku a nie np. wywołania procedury;
6. taki skok może prowadzić na zewnątrz aktualnie wykonywanych bloków, funkcji itd.;
7. miejsca docelowe tych skoków są łatwe do zlokalizowania w tekście programu;
8. skoki te powiny być wykonane *implicite* tak, by każdej akcji, która może się nie powieść nie trzeba było jawnie dozorować odpowiednią instrukcją warunkową i tak, by na wczesnym etapie programowania można było nie uwzględniać obsługi sytuacji awaryjnych a późniejsze ich dopisanie nie wymagało istotnych przeróbek w już istniejącym kodzie.

Eleganckim rozwiązaniem problemu obsługi sytuacji awaryjnych są *wyjątki* (ang. *exceptions*).[5] Wyjątki pozwalają na wprowadzenie do języka strukturalnej instrukcji skoku nowego rodzaju, zwanej instrukcją **raise** (Ada, SML), `throw` (C++) lub **signal** (CLU). Miejsca

[5] „Małe hałaśliwe stworzonka” (Oskar Miś)

docelowe takich skoków są łatwe do zlokalizowania w programie, instrukcje te są zatem dosyć bezpieczne i nie przyczyniają się do powstawania błędów w programie. Ich wystąpienie w programie nie zaburza też jego strukturalnej budowy.

Wyjątki opiszemy na przykładzie języka Ada, który jest jednym z pierwszych języków wyposażonym w taki mechanizm.

6.4.1. Wyjątki w Adzie

Sytuacje wyjątkowe mają swoje nazwy. Do nazywania wyjątków używa się w Adzie identyfikatorów. Pewnien zestaw wyjątków jest zdefiniowany w standardzie języka, np. NUMERIC_ERROR jest wyjątkiem opisującym błąd wykonania operacji arytmetycznej (dzielenie przez zero, nadmiar itp). Programista może tworzyć własne nazwy sytuacji wyjątkowych. Nowe wyjątki wprowadza się tak, jakby były zmiennymi specjalnego typu **exception** przy pomocy deklaracji:

$$E : \textbf{exception};$$

gdzie E jest nazwą deklarowanego wyjątku.

Jeżeli na skutek zaistnienia sytuacji awaryjnej wykonanie normalnego ciągu instrukcji nie może być kontynuowane, to można je przerwać za pomocą instrukcji *zgłoszenia wyjątku* postaci

$$\textbf{raise}\ E;$$

Instrukcja **raise** jest instrukcją skoku i przenosi sterowanie w inne miejsce programu.

Wyjątki można *obsługiwać* na końcu każdego bloku. Ciąg instrukcji, w trakcie wykonania którego mogą być zgłoszone wyjątki obejmuje się słowami kluczowymi **begin** i **end**, zaś przed słowem **end** wprowadza się tzw. *strefę wyjątków bloku* słowem **exception**. Strefa wyjątków jest ciągiem *segmentów obsługi wyjątków* postaci

$$\textbf{when}\ E \Rightarrow C;$$

gdzie E jest nazwą wyjątku, zaś C ciągiem instrukcji do wykonania w razie *przechwycenia* wyjątku E. Wyjątek E gra rolę etykiety, do której następuje skok wywołany instrukcją **raise**. W programie może być jednak wiele segmentów obsługi tego samego wyjątku. Miejsce obsługi wyjątku ustala się znajdując najmniejszy blok, który swoim zasięgiem obejmuje miejsce zgłoszenia wyjątku i który zawiera segment obsługi tego wyjątku. Po zlokalizowaniu odpowiedniego segmentu obsługi jest wykonywana jego treść a następnie sterowanie jest przekazane do instrukcji następującej bezpośrednio po danym bloku.

Jeżeli nie rozważamy procedur, to miejsce obsługi wyjątku jawnie zgłoszonego instrukcją **raise** można statycznie wyznaczyć na podstawie struktury programu. Instrukcja **raise** działa wówczas niemal tak samo, jak instrukcja skoku **goto**. Jednak miejsce obsługi wyjątku zgłoszonego w treści procedury ustala się biorąc pod uwagę miejsce jej *użycia* a nie *deklaracji*. Dlatego w czasie kompilacji nie można ustalić miejsca obsługi takiego wyjątku.

Przykładowy blok w Adzie, w którym użyto wyjątków jest przedstawiony w tablicy 6.4 na następnej stronie. Na jego początku jest zadeklarowany nowy wyjątek ZA_MALO. W treści bloku występują dwie instrukcje. W pierwszej z nich sprawdza się, czy X jest ujemne

```
declare
    ZA_MALO : exception;
begin
    if X < 0.0
        then raise ZA_MALO;
        else X := X / Y;
    end if;
    X:=X+1.0
exception
    when NUMERIC_ERROR ⇒
        X := 1.0;
    when ZA_MALO ⇒
        X := 0.0;
end
```

Tablica 6.4. Przykład programu z użyciem wyjątków w Adzie

i jeśli tak, zostaje zgłoszony wyjątek ZA_MALO, w przeciwnym zaś razie X jest dzielone przez Y. Jeżeli Y będzie równe zeru lub zbyt małe co do wartości bezwzględnej, to podczas dzielenia może być zgłoszony wyjątek NUMERIC_ERROR. Jeżeli nie zostanie on zgłoszony, to X jest zwiększane o jeden i wykonanie bloku zostaje zakończone. Jeżeli wykonanie treści bloku zostało przerwane na skutek zgłoszenia wyjątku, to odpowiedni wyjątek jest poszukiwany w strefie wyjątków bloku. Jeśli więc został zgłoszony wyjątek ZA_MALO, to zostanie wykonana instrukcja $X := 1.0$. Gdy np. Y jest równe zeru i wystąpił błąd dzielenia, wówczas zostanie wykonana instrukcja $X := 0.0$. Jeżeli w trakcie wykonania treści bloku zostanie zgłoszony inny wyjątek, niż ZA_MALO i NUMERIC_ERROR, segment jego obsługi będzie poszukiwany w dalszych blokach obejmujących swoim zasięgiem miejsce jego zgłoszenia. Jest to tzw. *propagacja wyjątku*. Nigdzie nie obsłużony wyjątek powoduje zerwanie działania programu.

W segmencie obsługi wyjątku może wystąpić instrukcja **raise**; bez parametru. Powoduje ona ponowne zgłoszenie wyjątku, który spowodował przekazanie sterowania do tego segmentu obsługi. Pozwala to na obsługe sytuacji awaryjnej „na raty”.

6.4.2. Wyjątki w SML-u

Wyjątki, ich zgłaszanie, deklarowanie i obsługa. W SML-u sytuacje wyjątkowe również mają swoje nazwy. Podobnie jak w Adzie, użytkownik może definiować nowe wyjątki deklaracją postaci

`exception` E

gdzie E jest nową nazwą wyjątku, na przykład:

```
- exception Katastrofa;
```

```
exception Katastrofa
```

System odpowiada, że identyfikator `Katastrofa` jest od teraz nazwą wyjątku. Wyjątki to wartości specjalnego typu `exn` i zachowują się tak, jak wartości wszelkich innych typów:

```
- Katastrofa;
val it = Katastrofa(-) : exn
- fun f () = Katastrofa;
val f = fn : unit -> exn
```

a ponieważ są *konstruktorami* typu `exn`, mogą pojawiać się we wzorcach, np.

```
- fun decyduj Katastrofa = "mój"
=   | decyduj _ = "nie mój";
val decyduj = fn : exn -> string
```

Status identyfikatora `Katastrofa` jest zatem podobny do statusu np. konstruktora listy pustej `nil`. Identyfikatory wyjątków pochodzą więc z tej samej przestrzeni nazw, co nazwy wartości.

Ciekawą własnością wyjątków, odróżniającą je od wartości wszelkich innych typów, jest możliwość ich *zgłoszenia*. Służy do tego specjalne wyrażenie postaci

$$\texttt{raise}\ M$$

gdzie M jest wyrażeniem typu `exn` którego wartością jest pewny (zadeklarowany wcześniej i widoczny w zasięgu powyższego wyrażenia) wyjątek E. Wyrażenie `raise` M nie posiada żadnej wartości i może wystąpić w dowolnym kontekście, a jego obliczenie powoduje przerwanie obliczeń i *zgłoszenie* wyjątku E. Dla przykładu funkcja `hd` ujawniająca głowę listy nie może wyznaczyć swojej wartości, gdy zostanie zaaplikowana do listy pustej (bo lista pusta nie ma głowy!). Deklarujemy zatem wyjątek `Hd`. Utarło się, że nazwy wyjątków zgłaszanych przez funkcje różnią się od nazw funkcji jedynie tym, że są pisane wielką literą (muszą być w jakiś sposób inne, bo żyją w tej samej przestrzeni nazw i inaczej by się wzajemnie przesłaniały). Jest to oczywiście jedynie zwyczaj, który można ignorować, jeśli znajdziemy lepszą nazwę dla nowego wyjątku. Funkcja `hd` dla listy pustej nie dostarcza żadnej wartości, tylko zgłasza wyjątek:

```
exception Hd
fun hd (x::_) = x
  | hd nil = raise Hd
```

W dowolnym miejscu programu możemy *obsłużyć* wyjątek używając wyrażenia postaci:

$$\begin{aligned} M\ \texttt{handle}\ E_1 &\texttt{=>}\ N_1\ \texttt{|} \\ E_2 &\texttt{=>}\ N_2\ \texttt{|} \\ &\vdots \\ E_n &\texttt{=>}\ N_n \end{aligned}$$

gdzie M jest wyrażeniem, w czasie obliczania którego mogą zostać zgłoszone wyjątki, N_i są zastępczymi wyrażeniami obliczanymi podczas obsługi wyjątków, a E_i to wzorce, do których wyjątki się dopasowują. Wzorcami mogą być nazwy wyjątków, zmienne (które dopasowują się do wszystkich wyjątków), zmienna anonimowa „_” albo wzorce dla wyjątków z parametrami, o czym powiemy dalej. Obliczenie powyższego wyrażenia polega na obliczeniu wyrażenia M. Jeśli zakończy się ono pomyślnie, otrzymany wynik jest wynikiem całego wyrażenia. Jeśli jednak obliczenie wyrażenia M zostanie zerwane na skutek zgłoszenia wyjątku E, zamiast wyrażenia M zostaje obliczone wyrażenie

```
case E of
    E1 => N1 |
    E2 => N2 |
        ⋮
    En => Nn |
     x => raise x
```

Jeśli dla wyjątku E nie przewidziano obsługi, zostanie ponownie zgłoszony. Jest to tzw. *propagacja* wyjątku. Będzie się on wówczas przesuwał wyżej w drzewie rozbioru programu, szukając kolejnych wyrażeń `handle`, które mogłyby go obsłużyć. Nigdzie nie obsłużony wyjątek E powoduje całkowite zerwanie obliczeń, wypisanie na konsoli komunikatu

`uncaught exception` E

i kolejnego znaku zachęty „-” w systemie interakcyjnym.

Deklaracje wyjątków, podobnie jak wszelkie inne deklaracje, można łączyć słowem `and`. Można dzieki temu zaoszczędzić nieco miejsca, szczególnie że `exception` jest jednym z najdłuższych słów kluczowych SML-a!
Szczególną uwagę należy zwrócić na kolejność rozbioru gramatycznego wyrażenia `handle`. W wyrażeniu

`if` B `then` M `else` N `handle ...`

obsłużone zostaną jedynie wyjątki zgłoszone w wyrażeniu N. Jeśli chcielibyśmy dozorować całe wyrażenie warunkowe, należy je otoczyć nawiasami:

`(if` B `then` M `else` N`) handle ...`

Wyrażenie

```
fn (x,0) => hd x handle Hd => 0 | _ => 1
```

oznacza

```
fn (x,0) => (hd x handle Hd => 0 | _ => 1)
```

Inny rozbiór musimy wymusić nawiasami:

```
fn (x,0) => (hd x handle Hd => 0) | _ => 1
```

(zauważmy, że to są dwie zupełnie różne funkcje). Zatem z `handle` w SML-u występuje podobny problem, jak z `case`. Uniknięto by tego, gdyby wyrażenie `handle` kończyło się słowem `end`.

Zen a sztuka oporządzania wyjątków. Zgłoszony wyjątek może niekiedy przeżyć swoją nazwę, np.

```
let exception E in raise E end
```

Zgłoszony wyjątek E jest propagowany na zewnątrz wyrażenia let, tj. na zewnątrz obszaru widoczności identyfikatora E. W wyrażeniu handle nie można zatem użyć jego nazwy. Nie oznacza to jednak, że nie można go obsłużyć — wzorzec w postaci pojedynczej zmiennej dopasowuje się do każdego wyjątku:

```
let exception E in raise E end handle _ => "Wszystko w porzadku!"
```

Jeśli napisaliśmy duży program prog, w którym zdarzyć się mogą różne nieprzewidziane sytuacje i często nawet nie wiadomo, jakie jeszcze wyjątki mogą zostać zgłoszone, możemy napisać:

```
prog () handle x => (print "Internal software error. Sorry!\n";
                     raise x)
```

W razie zgłoszenia kompletnie nieprzewidzianego wyjątku obliczenia zostaną i tak zerwane, ale wcześniej system wypisze w miarę zrozumiały komunikat dla użytkownika. Zawsze to robi lepsze wrażenie (to jest bardzo psychologiczne — użytkownik ma wrażenie, że mimo iż program zawiera błędy, to sytuacja jest pod kontrolą).

Wyrażeniem handle warto otaczać zamknięte fragmenty programu, przy awarii których można podjąć jakąś sensowną akcję zastępczą. Dlatego nie ma sensu np. dozorowanie każdej operacji wejścia/wyjścia. Lepiej pozwolić wyjątkowi na propagację aż do miejsca wywołania całego podprogramu obsługi plików i tam zaprogramować np. awaryjny zapis do plików tymczasowych, gdy główne procedury wypisywania wyników z jakichś powodów zawiodą.

Niekiedy wyjątków używa się do innych celów niż zrywanie obliczeń na skutek błędu. W programach rekurencyjnych zgłoszenie wyjątku pozwala na natychmiastowe wyjście z zagłębienia rekurencyjnego i nie trzeba się wycofywać krok po kroku. Jeśli w trakcie obliczeń poznamy już wynik, można też za pomocą wyjątku zerwać obliczenia, np.

```
fun prod xs =
   let
     exception Aux
   in
     foldr (fn (x,p) => if x=0 then raise Aux else x*p) 1 xs
        handle Aux => 0
   end
```

Funkcja prod oblicza iloczyn elementów listy liczb całkowitych. Jeżeli podczas obliczeń natrafimy na liczbę 0, dalsze mnożenie jest niepotrzebne i jako wynik natychmiast możemy zwrócić wartość 0. W tym celu obliczenie jest zerwane za pomocą zgłoszenia wyjątku Aux. Pomimo pewnej użyteczności takich sztuczek, nie należy ich nadużywać w programowaniu, zwłaszcza że powyższą funkcję można elegancko zaprogramować w następujący sposób:

```
val prod =
   let
      fun aux  _  (0::_)  = 0
        | aux acc (x::xs) = aux (x*acc) xs
        | aux acc   nil   = acc
   in
      aux 1
   end
```

Wyjątki z parametrami. Często wyjątek nie niesie ze sobą wystarczająco wiele informacji, np. przy zaistnieniu błędu operacji wejścia/wyjścia chcielibyśmy, by razem z wyjątkiem wędrował komunikat o tym, co się właściwie stało, w postaci np. łańcucha znaków. W SML-u przewidziano taką możliwość — konstruktor wyjątku może mieć parametr. Wyjątki z parametrami wprowadzamy deklaracją

`exception` E `of` σ

gdzie E jest jak poprzednio nazwą wyjątku, a σ — typem parametru. E jest wówczas typu $\sigma \rightarrow$ `exn`. Np.

```
exception ZaMalo of int
exception SyntaxError of {line : int, column : int}
```

Aby zgłosić wyjątek z parametrem, należy najpierw zaaplikować go do wartości odpowiedniego typu, np.

```
raise ZaMalo 500
raise SyntaxError {line=20, column=5}
```

Słowo `raise` jest słowem kluczowym, nie funkcją, nawiasy wokół aplikacji wyjątku do parametru są więc zbyteczne. Wzorzec dla wyjątku z parametrem ma postać $E\ P$, gdzie E jest nazwą wyjątku, zaś P — dowolnym wzorcem typu σ, np.

```
exception E of int list
fun eval f x = f x
      handle E (x::xs) => 0
           | E   nil   => 1
```

Wyjątki standardowe. W standardzie SML-a przewidziano dwa predefiniowane wyjątki:

```
exception Match and Bind
```

Wyjątki te są zgłaszane podczas dopasowania wzorca odpowiednio w wyrażeniu `case` i definicji funkcji oraz w deklaracji `val` (może się to zdarzyć tylko wówczas, gdy wzorce nie są wyczerpujące), np.

```
- val (x::xs) = nil : int list;
Warning: binding not exhaustive
       x :: xs = ...
uncaught exception nonexhaustive binding failure
- (fn 1 => 1) 2;
Warning: match nonexhaustive
          1 => ...
uncaught exception nonexhaustive match failure
```

System SML/NJ ostrzega przed takimi sytuacjami (SML/NJ nawet w głównej pętli interpretera, co jest niezgodne ze standardem). W SML/NJ zdefiniowano dodatkowo wyjątek

```
exception Fail of string
```

obsługiwany przez główną pętlę interpretera, która wypisuje na konsoli jego argument. Wygodny szczególnie podczas uruchamiania programu.

6.4.3. Wyjątki w C++

W języku C++ wyjątki są wartościami dowolnego typu, najczęściej obiektami specjalnej klasy (o obiektach mówimy w rozdziale 17). Wyjątki bez parametrów wprowadza się definiując klasę „pustych" obiektów, np.

```
class za_malo {};
```

Wyjątki zgłasza się przy pomocy *wyrażenia zgłoszenia wyjątku* postaci:

`throw` E`;`

gdzie E jest wyrażeniem dostarczającym wyjątku, np. `za_malo()` jest wyrażeniem dostarczającym nowego obiektu klasy `za_malo`, zatem wyjątek `za_malo` można zgłosić następująco:

```
throw za_malo();
```

Do obsługi wyjątków służą instrukcje zwane *blokiem* `try` i *blokiem* `catch`. Blok `try` może wystąpić w dowolnym miejscu programu, w którym może wystąpić instrukcja. Bezpośrednio za nim może wystąpić ciąg bloków `catch`:

`try {` C `}`
`catch (`E_1`) {` C_1 `}`
`catch (`E_2`) {` C_2 `}`
$\vdots$
`catch (`E_n`) {` C_n `}`

gdzie E_i są deklaracjami wyjątków, zaś C i C_i są ciągami instrukcji. Jeżeli wykonanie ciągu instrukcji C zostanie zerwane na skutek zgłoszenia wyjątku E_i, wówczas w zamian zostaną wykonane instrukcje C_i.

Odpowiednik SML-owych wyjątków z parametrami osiaga się deklarując wyjątki jako klasę, której pola są parametrami wyjątku lub jako typ, który posiada wiele elementów, np.

```
enum BLAD_ZAKRESU {ZA_DUZO, ZA_MALO};
   ...
try {
   ...
}
catch (BLAD_ZAKRESU b) {
   switch (b) {
      case ZA_DUZO: printf ("za dużo"); break;
      case ZA_MALO: printf ("za mało"); break;
   }
}
```

Podobnie jak w Adzie, w treści procedury obsługi wyjątku może wystąpić wyrażenie zgłoszenia wyjątku `throw` bez parametru. Powoduje ono ponowne zgłoszenie wyjątku, który spowodował przekazanie sterowania do tej procedury obsługi.

6.4.4. Bloki finally w Javie

Wyjątki w Javie są zorganizowane podobnie, jak w języku C++, nie będziemy ich więc tu dokładnie opisywać (zwłaszcza, że wymagałoby to wyjaśnienia wielu szczegółów dotyczących obiektów, o których mówimy w rozdziale 17). Warto jedynie wspomnieć o tzw. *blokach* `finally`. Umieszcza się w nich czynności, które powinny być wykonane na końcu pewnego fragmentu programu niezależnie od tego, czy wykonanie tego fragmentu powiodło się, czy nie. Bezpośrednio po bloku `try` i blokach `catch` może w Javie wystąpić blok `finally` postaci

`finally` { *C* }

Instrukcje *C* będą wykonane niezależnie od tego, w jaki sposób zakończyło się wykonanie bloków `try` i `catch`. Na przykład jeśli w programie

```
if (datafile.open(filename)) {
   try {
      while((line=datafile.readLine()) != null)
         i++;
   }
   catch (IOException e) {
      handleIOExceptions(e);
   }
   finally {
      datafile.close();
   }
}
```

powiedzie się otwarcie pliku, to rozpocznie się wczytywanie jego kolejnych wierszy w pętli `while`. Niezależnie od tego, czy zakończy się ono pomyślnie, czy też zostanie zgłoszony (i być może obsłużony) wyjątek, na końcu zostanie wykonana metoda `close` zamykająca plik.

6.5. Instrukcje strukturalne w językach programowania

Projektując język programowania należy rozstrzygnąć, jakie instrukcje sterujące powinny się w nim znaleźć. Z jednej strony język wyposażony w dużą liczbę barokowych konstrukcji składniowych (np. PL/I, od dawna już nie używany) jest trudny do opanowania, programy są trudne do zrozumienia, a i twórcom kompilatorów nie ułatwia to pracy. Z drugiej strony zbyt mała liczba dostępnych instrukcji może utrudniać zwięzłe i czytelne zapisywanie programu.

6.5.1. Instrukcja złożona

Instrukcja złożona to konstrukcja składniowa pozwalająca na potraktowanie ciągu instrukcji jako pojedynczej instrukcji. W tym celu otacza się ciąg instrukcji parą słów kluczowych **begin** i **end** (Pascal, Ada), BEGIN i END (Modula 2), { i } (C, C++, Java). Niekiedy używa się innych znaków, np. (i) (Algol 68, język D). Instrukcji złożonej używa się często wówczas, gdy zachodzi potrzeba umieszczenia ciągu instrukcji w kontekście, w którym składnia języka wymaga umieszczenia pojedynczej instrukcji (np. fraza **then** lub treść instrukcji pętli w Pascalu). Koncepcja grupowania instrukcji stanowi podstawę programowania strukturalnego. Fragment programu otoczony słowami **begin** i **end**, jeżeli nie posiada dodatkowych wejść (etykiet) i wyjść (skoków), może być traktowany jako zamknięta całość, „czarna skrzynka".

Jeżeli instrukcje strukturalne są zakończone jawnymi terminatorami (Ada, Modula 2), wówczas składnia dopuszcza umieszczenie w ich wnętrzu ciągu instrukcji i wówczas instrukcja złożona jest z nimi zintegrowana i nie trzeba jej używać *explicite*.

W większości języków (wyjątkiem jest Pascal) wariantem instrukcji złożonej jest *blok*, tj. instrukcja złożona na początku (lub w treści) której można umieszczać deklaracje zmiennych (mówimy o tym w rozdziale 7).

Uwagi. Z dzisiejszej perspektywy może się to wydawać zaskakujące, że np. w FORTRAN-ie 66 w treści logicznej instrukcji warunkowej IF (b) I mogła wystąpić pojedyncza instrukcja. Aby warunkowo wykonać ciąg instrukcji należało użyć instrukcji skoku. Instrukcja złożona po raz pierwszy pojawiła się w języku Algol 60.

6.5.2. Instrukcje wyboru sterowane wyrażeniem logicznym

W większości języków programowania jest dostępna instrukcja warunkowa postaci

$$\textbf{if } b \textbf{ then } \vec{I} \textbf{ end}$$

również w wersji z frazą **else**:

$$\textbf{if } b \textbf{ then } \vec{I}_1 \textbf{ else } \vec{I}_2 \textbf{ end}$$

gdzie b jest wyrażeniem logicznym a $\vec{I}$, $\vec{I_1}$ i $\vec{I_2}$ są ciągami instrukcji. Jeśli należy sprawdzić wiele warunków i podjąć jedną z wielu czynności, wówczas zapisuje się zwykle kaskadę instrukcji **if** (nie jest to ich zagnieżdżenie, ponieważ kolejna instrukcja **if** występuje w gałęzi **else** poprzedniej instrukcji). Słowo kluczowe **end** (terminator instrukcji), jeśli występuje, znacznie ułatwia jednoznaczny rozbiór gramatyczny programu (była o tym mowa w rozdziale 3.10.3), jednak wówczas na końcu kaskady instrukcji **if** pojawia się długi ciąg słów **end**. Ponieważ taki zapis jest mało czytelny, w niektórych językach zamiast jednego warunku może być ich w instrukcji **if** wiele, a pary postaci *warunek* **then** *instrukcja* są oddzielane od siebie dodatkowym słowem kluczowym **elif**, ELSIF itp. Schemat tak rozbudowanej instrukcji warunkowej w Adzie i Moduli 2 jest przedstawiony na rysunku 6.8 na następnej stronie.

Uwagi. Instrukcja warunkowa występuje nawet w najstarszych językach programowania. Dawniej była zwykle kojarzona z instrukcją skoku, np. w postaci tzw. instrukcji *arytmetycznego IF* w FORTRAN-ie 66:

$$\text{IF } (e)\ l_1, l_2, l_3$$

gdzie e jest wyrażeniem arytmetycznym, zaś l_1, l_2 i l_3 trzema etykietami. Wykonanie takiej instrukcji polega na wykonaniu skoku do etykiety l_1, gdy wartość wyrażenia e jest ujemna, l_2, gdy wyrażenie e ma wartość 0 i l_3, gdy wartość wyrażenia e jest dodatnia. Jeśli nie była skojarzona z instrukcją skoku, pozwalała zwykle jedynie na warunkowe wykonanie pojedynczej instrukcji (bez możliwości wykonania ciągu instrukcji) i nie posiadała frazy ELSE, np. tak jest zbudowana tzw. instrukcja *logicznego IF* w FORTRAN-ie 66 postaci

$$\text{IF } (b)\ I$$

gdzie b jest wyrażeniem logicznym, zaś I pojedynczą instrukcją prostą. We współczesnej postaci (z gałęzią **else** i z możliwością warunkowego wykonania ciągu instrukcji) instrukcja warunkowa pojawiła się po raz pierwszy w języku Algol 60.

6.5.3. Instrukcje wyboru sterowane wyrażeniem arytmetycznym

Niekiedy wyboru jednej z wielu możliwych czynności jest wygodnie dokonać na podstawie wartości wyrażenia np. całkowitoliczbowego a nie logicznego. W Pascalu do tego celu służy instrukcja **case** przedstawiona na rysunku 6.9 (a), w której w jest wyrażeniem sterującym typu porządkowego, zaś $e_1^i, \ldots, e_{k_i}^i$, dla $i = 1, \ldots, n$, jest ciągiem $k_i \geq 1$ oddzielonych przecinkami stałych tego typu. Stałe wyboru występujące w jednej instrukcji **case** powinny być parami różne. W instrukcji **case** dokonuje się wyboru jednej z n możliwości, zależnie od wartości wyrażenia sterującego. Brak stałej wyboru odpowiadającej obliczonej wartości wyrażenia sterującego jest błędem i powoduje przerwanie obliczeń. Schemat blokowy instrukcji **case** jest podobny do schematu instrukcji **if** z tą różnicą, że zamiast dwóch może wystąpić wiele gałęzi. Instrukcja **case** jest zatem instrukcja strukturalną. Zauważmy, że instrukcja warunkowa

$$\textbf{if } b \textbf{ then } I_1 \textbf{ else } I_2$$

jest równoważna instrukcji

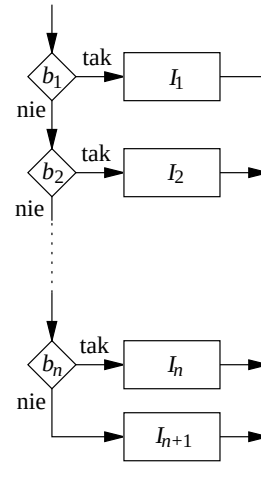
$$\textbf{case } b \textbf{ of } \mathit{true} : I_1;\ \mathit{false} : I_2 \textbf{ end}$$

Ada:

if b_1 **then** I_1; **elif**
b_2 **then** I_2; **elif**
⋮
b_n **then** I_n;
else I_{n+1};
end if;

Modula 2:

IF b_1 THEN I_1 ELSIF
b_2 THEN I_2 ELSIF
⋮
b_n THEN I_n
ELSE I_{n+1}
END



Rysunek 6.8. Instrukcja warunkowa z wieloma warunkami w Adzie i Moduli 2

Z drugiej strony znaczenie instrukcji **case** można wyjaśnić sprowadzając ją do kaskady instrukcji **if** (zob. rysunek 6.9) po wprowadzeniu dodatkowej, nie występującej w programie zmiennej X, potrzebnej do przechowania obliczonej wartości wyrażenia sterującego.

W Adzie jest możliwe umieszczenie na końcu instrukcji **case** dodatkowej, opcjonalnej frazy **others**. Instrukcja występująca w gałęzi **others** jest wykonywana wówczas, gdy nie ma gałęzi odpowiadającej obliczonej wartości wyrażenia. Opis zbioru wartości, jakie może przyjąć wyrażenie sterujące, by została wykonana dana gałąź instrukcji **case**, tzw. *ewentualność* lub *etykieta wyboru* może być w Adzie nie tylko stałą typu całkowitoliczbowego lub wyliczeniowego, lecz dowolnym wyrażeniem stałym (tj. takim, którego wartość można obliczyć w czasie kompilacji programu) dostarczającym takiej wartości lub zakresem całkowitym postaci $c..d$, gdzie c i d są wyrażeniami stałymi. Jedna gałąź może być etykietowana wieloma ewentualnościami oddzielonymi od siebie kreskami pionowymi |. Wszystkie ewentualności występujące w jednej instrukcji **case** muszą być rozłączne. Bardzo podobną postać, różniącą się od rozwiązania przyjętego w Adzie jedynie szczegółami składni ma instrukcja CASE w Moduli 2. Postać instrukcji **case** w Adzie i CASE w Moduli 2 jest przedstawiona w tablicy 6.5 na stronie 140, w której e^i_j są ewentualnościami, a $\vec{I}_i$ ciągami instrukcji.

W języku C występuje, z pozoru podobna do adowej instrukcji **case**, instrukcja `switch` postaci

`switch (`w`)` I

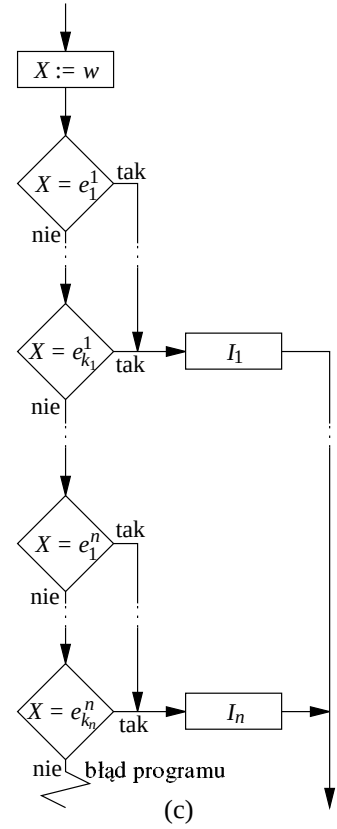
Wyrażenie sterujące w powinno być typu całkowitoliczbowego, I zaś jest instrukcją, przeważnie blokiem, w którym występują instrukcje etykietowane wartościami typu całkowitoliczbowego. Etykiety mają postać `case` e`:`, gdzie e jest wyrażeniem stałym typu całko-

case w **of**
$e_1^1, \ldots, e_{k_1}^1 : I_1;$
$\vdots$
$e_1^i, \ldots, e_{k_i}^i : I_i;$
$\vdots$
$e_1^n, \ldots, e_{k_n}^n : I_n$
end

(a)

$X := w;$
if $X = e_1^1$ **or** $\ldots$ **or** $X = e_{k_1}^1$ **then** I_1 **else**
$\vdots$
if $X = e_1^i$ **or** $\ldots$ **or** $X = e_{k_i}^i$ **then** I_i **else**
$\vdots$
if $X = e_1^n$ **or** $\ldots$ **or** $X = e_{k_n}^n$ **then** I_n **else**
halt ('brak odpowiedniej stałej wyboru')

(b)



(c)

Rysunek 6.9. (a) Instrukcja **case** w Pascalu, (b) jej przekład na kaskadę instrukcji **if** i (c) odpowiadający temu przekładowi schemat blokowy

witoliczbowego (zob. rysunek 6.4 na stronie 118). Wszystkie etykiety występujące w zasięgu instrukcji `switch` muszą być parami różne. Dodatkowo etykietą może być `default:`. Ta etykieta może wystąpić w dowolnym miejscu, niekoniecznie jako ostatnia w instrukcji `switch`, jednak co najwyżej raz. Wykonanie instrukcji `switch` polega na obliczeniu wyrażenia sterującego *w*, po czym następuje skok do instrukcji o etykiecie będącej wyliczoną wartością wyrażenia sterującego. Jeżeli taka etykieta nie istnieje, wykonuje się skok do instrukcji opatrzonej etykietą `default:` (jeśli występuje), lub przechodzi się do wykonania instrukcji następującej po instrukcji `switch`. Zauważmy, że instrukcja `switch` *de facto* nie jest instrukcją strukturalną, jej ciało posiada bowiem wiele wejść (tyle, ile jest etykiet). Po wykonaniu instrukcji, do której nastąpił skok wykonuje się bowiem kolejne, następujące po niej instrukcje, nawet jeśli posiadają inne etykiety `case` i `default`. W celu przekazania sterowania na koniec instrukcji `switch` należy użyć instrukcji skoku `break`. Instrukcja `switch` jest zatem instrukcją „niższego poziomu" niż instrukcja **case** w Pascalu i należy w istocie do

Ada:	Modula 2:
case w **is**	CASE w OF
when $e_1^1 \mid \ldots \mid e_{k_1}^1 \Rightarrow \vec{I}_1$	$e_1^1, \ldots, e_{k_1}^1 : \vec{I}_1 \mid$
$\vdots$	$\vdots$
when $e_1^i \mid \ldots \mid e_{k_i}^i \Rightarrow \vec{I}_i$	$e_1^i, \ldots, e_{k_i}^i : \vec{I}_i \mid$
$\vdots$	$\vdots$
when $e_1^n \mid \ldots \mid e_{k_n}^n \Rightarrow \vec{I}_n$	$e_1^n, \ldots, e_{k_n}^n : \vec{I}_n$
when others $\Rightarrow \vec{I}_{n+1}$	ELSE $\vec{I}_{n+1}$
end case;	END;

Tablica 6.5. Postać instrukcji **case** w Adzie i CASE w Moduli 2

grupy strukturalnych instrukcji skoku.

Uwagi. Instrukcja `switch` w języku C powstała w wyniku „ustrukturalizowania" niezwykle niebezpiecznej instrukcji skoku języka Algol 60. Otóż parametrem d instrukcji

goto d

w Algolu 60 może być nie tylko etykieta, ale znacznie ogólniejsze *wyrażenie desygnujące*, tj. wyrażenie, którego wartością jest etykieta. Składnia wyrażeń desygnujących jest następująca:

$$d \quad ::= \quad l \mid (d) \mid \textbf{if } b \textbf{ then } d \textbf{ else } d \mid S[e]$$

gdzie l jest etykietą, b wyrażeniem logicznym, e wyrażeniem arytmetycznym, zaś S tzw. *przełącznikiem*, wprowadzanym deklaracją postaci

$$\textbf{switch } S := d_1, \ldots, d_n;$$

gdzie $d_1, \ldots, d_n$ są wyrażeniami desygnującymi. W zasięgu powyższej deklaracji przełącznika wartością wyrażenia desygnującego $S[e]$ jest wartość wyrażenia d_i, jeśli bieżącą wartością wyrażenia arytmetycznego e jest i. Jeżeli wartość wyrażenia e jest mniejsza niż 1 bądź większa niż n, wartość wyrażenia desygnującego jest nieokreślona. Wykonanie instrukcji **goto** d, gdy wartość wyrażenia d jest nieokreślona powoduje przejście do wykonania następnej instrukcji.

Jeżeli $l_1, \ldots, l_n$ są etykietami w programie algolowym i chcemy przekazać sterowanie do instrukcji o etykiecie l_{k_i}, jeśli wartością pewnego wyrażenia e jest i, to postać programu będzie taka, jak przedstawiona w tablicy 6.6 na następnej stronie, gdzie $\vec{I}_j$ są ciągami instrukcji. W tym przypadku definicja przełącznika zawiera przeważnie stałe etykiety, a nie dowolne wyrażenia desygnujące, a obszar skoków jest ograniczony do jednego, niewielkiego bloku. Czytelność programu można ponadto znacznie zwiększyć, jeśli zamiast przełącznika i zwykłych etykiet wprowadzi się specjalne etykiety, zawierające stałe całkowite. Otrzymamy w ten sposób instrukcję `switch` języka C.

Wyrażenia desygnujące pozwalają tworzyć namiastki procedur za pomocą tzw. *skoków ze śladem*. Jeżeli wydzielimy pewien ciąg instrukcji $\vec{I}$ jako osobny podprogram i zechcemy go wykonać w n różnych miejscach programu, to możemy pierwszą instrukcję z ciągu $\vec{I}$ opatrzyć etykietą, np. p i w n miejscach programu umieścić instrukcję **goto** p. Po wykonaniu ciągu instrukcji $\vec{I}$ sterowanie musi jednak powrócić do miejsca, skąd nastąpił skok. Musi zatem istnieć pewna zmienna, np. *ret* (zwana niekiedy *adresem powrotu*), która będzie przechowywać informację o miejscu w programie, z którego przekazano sterowanie do podprogramu $\vec{I}$. Skok do tego miejsca może być następnie zrealizowany przy użyciu przełącznika:

begin	
switch $S := l_1, l_2, \ldots, l_n$;	`switch (`e`) {`
goto $S[e]$;	`default:` $\vec{I}_0$
$\vec{I}_0$	`case` k_1`:` $\vec{I}_1$
l_{k_1}: $\vec{I}_1$	`case` k_2`:` $\vec{I}_2$
l_{k_2}: $\vec{I}_2$	$\vdots$
$\vdots$	`case` k_n`:` $\vec{I}_n$
l_{k_n}: $\vec{I}_n$	`}`
end	

Tablica 6.6. Typowa postać programu w Algolu 60 wykorzystującego przełącznik i odpowiadający mu schemat instrukcji `switch` w języku C

switch $S := l_1, \ldots, l_n$;
$\vdots$
ret $:= 1$; **goto** p;
$l_1 : \ldots$
ret $:= 2$; **goto** p;
$l_2 : \ldots$
$\vdots$
ret $:= n$; **goto** p;
$l_n : \ldots$

gdzie „podprogram” jest zdefiniowany następująco:

$$p : \vec{I}; \ \textbf{goto}\ S[ret];$$

Podobna, choć mniej ogólna instrukcja tzw. *skoku wyliczanego* występuje w języku FORTRAN 66 i ma następującą postać:

$$\text{GO TO } (l_1, \ldots, l_n)\ N$$

w której N jest całkowitoliczbową *zmienną sterującą*. Wykonanie takiej instrukcji powoduje skok do instrukcji o etykiecie l_i, jeśli bieżącą wartością N jest i lub przejście do wykonania następnej instrukcji, jeśli i nie zawiera się w przedziale $1, \ldots, n$.

Rozwój instrukcji skoku od prostej instrukcji skoku do podanej stałej etykiety w programie poprzez instrukcję skoku wyliczanego osiągnął swoją kulminację w niezwykle ogólnej koncepcji wyrażenia desygnującego w Algolu 60. Na skutek problemów, jakie powodują instrukcje skoku i rozwoju metodologii programowania strukturalnego opisane tu instrukcje sterujące wyszły szybko z użycia.

Strukturalna instrukcja **case** pojawiła się po raz pierwszy w Algolu 68 i ma w nim postać

$$\textbf{case}\ e_1\ \textbf{in}\ I_1, \ldots, I_k\ \textbf{out}\ I_{k+1}\ \textbf{esac}$$

Jej wykonanie polega na obliczeniu wyrażenia e i następnie wykonaniu instrukcji I_j, jeżeli wartością wyrażenia e jest j, dla $j = 1, \ldots, n$ lub instrukcji I_{k+1}, jeżeli $j < 1$ lub $j > n$. Pełna składnia instrukcji **case** w Algolu 68 jest bardziej rozbudowana:

$$\textbf{case}\ e_1^1\ \textbf{in}\ I_1^1, \ldots, I_{k_1}^1\ \textbf{ouse}\ e_1^2\ \textbf{in}\ I_1^2, \ldots, I_{k_2}^2\ \ldots\ \textbf{ouse}\ e_1^n\ \textbf{in}\ I_1^n, \ldots, I_{k_n}^n\ \textbf{out}\ I^{n+1}\ \textbf{esac}$$

Słowo kluczowe **ouse** pełni podobną rolę jak **elif** w instrukcji warunkowej.

Instrukcja **case** we współczesnej postaci została wynaleziona przez C.A.R. Hoare’a. Po raz pierwszy pojawiła się w języku Algol W.

Do uzupełnienia:

6.5.4. Instrukcje powtarzania sterowane wyrażeniem logicznym
6.5.5. Instrukcje powtarzania sterowane wyrażeniem arytmetycznym
6.6. Przekład instrukcji strukturalnych na język maszyny

6.7. Zadania

Zadanie 6.1. Gratulacje! Właśnie zostałeś zatrudniony jako programista w firmie Macrohard produkującej oprogramowanie dla znanego i lubianego procesora Sextium II (opisanego w podrozdziale 4.1). Twoim zadaniem jest napisanie kompilatora skrośnego[6] języka D z podrozdziału 3.11 generującego kod wynikowy dla procesora Sextium II. Możesz wykorzystać parser z zadania 3.29.

[6]Kompilator skrośny to program działający na maszynie A generujący kod dla maszyny B. Twój kompilator będzie prawdopodobnie działał na komputerze PC lub podobnym, a będzie generował kod dla procesora Sextium II.

Rozdział 7

Procedury i rekursja

Do uzupełnienia:

7.1. Procedury
7.1.1. Przekazywanie sterowania do procedur
7.1.2. Konteksty lokalne: blokowa struktura programu; rekordy aktywacji i stos wywołań
7.1.3. Rekursja: linki sterowania, budowa rekordów aktywacji w języku C
7.1.4. Rekursja w kontekstach lokalnych: linki dostępu, budowa rekordów aktywacji w języku Pascal
7.1.5. Funkcje wyższych rzędów: *funarg problem*, *function closures*
7.1.6. Coroutines
7.2. Przekazywanie parametrów
7.2.1. Porządki wartościowania: wartościowanie gorliwe i leniwe
7.2.2. Przekazywanie parametrów w językach imperatywnych: przekazywanie parametrów przez nazwę (leniwe) i wartość (gorliwe); odmiany: przekazywanie parametrów przez zmienną (referencję) i przez wynik
7.2.3. Dynamiczny i statyczny zasięg zmiennych
7.2.4. Kolejność obliczania wyrażeń w języku SML
7.3. Zadania

Literatura zastępcza: [190, 101]

Rozdział 8

Elementy algebry abstrakcyjnej

Algebra abstrakcyjna (uniwersalna) jest treścią różnych wykładów podstawowych, m. in. „Wstępu do teorii mnogości i logiki" (zob. [173], str. 85–112). Dlatego autor zakłada, że prezentowane w bieżącym rozdziale definicje i twierdzenia są Czytelnikowi przynajmniej pobieżnie znane. Nowością mogą być pojęcia gatunku termu i algebry wielogatunkowej, będące jednak oczywistymi uogólnieniami pojęć, z którymi Czytelnik już się zetknął. Dlatego większość definicji jest bardzo zwięzła i pozbawiona komentarza, a dowody twierdzeń są pominięte lub jedynie naszkicowane.

Matematyka jest sztuką budowania abstrakcyjnych modeli rzeczywistości. Logika matematyczna zajmuje się badaniem sposobów opisu, mówienia i rozumowania na temat rzeczywistości. W świecie istnieją różne obiekty. Gdy staną się one przedmiotem naszego zainteresowania, zaczynamy o nich mówić. Matematycznymi modelami naszych wypowiedzi są *termy* (*wyrażenia*), a zbiorów obiektów, o których mówimy — *algebry*. W rozdziale 3 opisywaliśmy języki w sposób bardzo konkretny, jako ciągi znaków. Obecnie rozważymy metody ich abstrakcyjnego opisu.

8.1. Sygnatury i termy

Niech $\mathcal{S} \neq \emptyset$ będzie niepustym (przeważnie skończonym) zbiorem *gatunków* (*rodzajów*, ang. *sorts*). Jego elementy zwykle oznaczamy literami a, b itd, niekiedy z indeksami. Skończony niepusty ciąg gatunków $\langle a_1, \ldots, a_n, b\rangle$ dla $n \geq 0$ i $a_1, \ldots, a_n, b \in \mathcal{S}$ nazywamy *typem algebraicznym* (krócej *typem*), oznaczamy σ, τ, ρ itd, niekiedy z indeksami i zapisujemy w postaci $a_1 \times \ldots \times a_n \to b$ dla $n > 0$ oraz b dla $n = 0$. Liczbę n nazywamy *arnością* typu. Zbiór typów algebraicznych oznaczamy $\mathbb{T}_1(\mathcal{S})$. Z każdym typem τ związujemy zbiór Σ^τ *symboli typu* τ. Symbole typu τ oznaczamy f^τ, g^τ itd, niekiedy z indeksami. *Arnością* (*liczbą argumentów*) symbolu nazywamy arność jego typu. Symbole o arności 0, tj. typu $\tau = a \in \mathcal{S}$ nazywamy *stałymi* i oznaczamy c^a, d^a itd, niekiedy z indeksami. Symbole o arności 1 nazywamy *unarnymi*, symbole o arności 2 zaś *binarnymi*. Z każdym gatunkiem $a \in \mathcal{S}$ związujemy (zwykle przeliczalny nieskończony) zbiór $\mathcal{X}^a$ *zmien-*

nych gatunku a. Zmienne gatunku a oznaczamy x^a, y^a, z^a itd, niekiedy z indeksami. Zakładamy, że zbiory Σ^τ i $\mathcal{X}^a$ są parami rozłączne, choć czasem dopuszczamy pewne wyjątki. Rodzinę $\Sigma = \{\Sigma^\tau\}_{\tau \in \mathbb{T}_1(\mathcal{S})}$ nazywamy *sygnaturą algebraiczną* (krócej *sygnaturą*), rodzinę $\mathcal{X} = \{\mathcal{X}^a\}_{a \in \mathcal{S}}$ zaś *rodziną zmiennych*. Gdy nie prowadzi to do nieporozumień, pomijamy oznaczenie typu lub gatunku i piszemy f, c, x zamiast f^τ, c^a, x^a. Piszemy też $x \in \mathcal{X}$ na oznaczenie faktu, że $x \in \mathcal{X}^a$ dla pewnego $a \in \mathcal{S}$ itp.

Zbiory *termów* (*wyrażeń*) gatunku $a \in \mathcal{S}$ nad sygnaturą Σ i zbiorem zmiennych $\mathcal{X}$, oznaczane $\mathcal{T}^a(\Sigma, \mathcal{X})$ dla $a \in \mathcal{S}$, definiujemy indukcyjnie:

1. każda zmienna gatunku a jest termem tego gatunku, tj. $\mathcal{X}^a \subseteq \mathcal{T}^a(\Sigma, \mathcal{X})$;
2. jeżeli $t_i \in \mathcal{T}^{a_i}(\Sigma, \mathcal{X})$ dla $i = 1, \ldots, n$ i $n \geq 0$ oraz $f \in \Sigma^{a_1 \times \ldots \times a_n \to b}$, to para złożona z symbolu f i ciągu termów $\langle t_1, \ldots, t_n \rangle$ jest termem gatunku b, tj.
$$\langle f, \langle t_1, \ldots, t_n \rangle \rangle \in \mathcal{T}^b(\Sigma, \mathcal{X})$$
3. każdy term można zbudować używając reguł 1 i 2.

Term $\langle f, \langle t_1, \ldots, t_n \rangle \rangle$ przeważnie zapisujemy w notacji prefiksowej z nawiasami, tj. w postaci $f(t_1, \ldots, t_n)$, choć dla niektórych symboli binarnych przyjmujemy notację infiksową (zob. podrozdział 3.2). Termy przedstawiamy także graficznie w postaci drzewa o wierzchołkach etykietowanych symbolami z sygnatury (zob. rysunek 8.1 na sąsiedniej stronie). Jeżeli c jest symbolem o arności 0 (stałą), to term złożony z tego symbolu zapisujemy po prostu jako c. Nie rozpatrujemy osobno przypadku symboli o arności 0. Pisząc „term $f(t_1, \ldots, t_n)$ dla $n \geq 0$" mamy na myśli term $f(t_1, \ldots, t_n)$ gdy $n > 0$ i term f, gdy $n = 0$. Podobnie $a_1 \times \ldots \times a_n \to b$ dla $n = 0$ oznacza typ b. Używając skróconego zapisu sumy zbiorów $\bigcup A_i$ przyjmujemy, że suma pustej rodziny zbiorów jest zbiorem pustym.

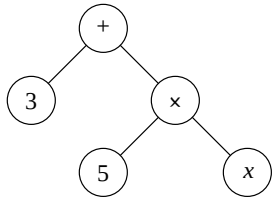
Przykład 8.1. Rozważmy zbiór gatunków $\mathcal{S} = \{N, B\}$, sygnaturę

$$\begin{aligned}
\Sigma^N &= \{0, 1, 2, \ldots\} \\
\Sigma^B &= \{T, F\} \\
\Sigma^{B \to B} &= \{\neg\} \\
\Sigma^{N \times N \to N} &= \{+, \times\} \\
\Sigma^{N \times N \to B} &= \{\leq, \geq, <, >, =, \neq\} \\
\Sigma^{B \times B \to B} &= \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}
\end{aligned}$$

przy czym zbiory Σ^τ dla pozostałych typów τ są puste, oraz zbiory zmiennych

$$\begin{aligned}
\mathcal{X}^N &= \{x, y, z, \ldots\} \\
\mathcal{X}^B &= \{p, q, r, \ldots\}
\end{aligned}$$

Przykładem termu gatunku N jest $+(3, \times(5, x))$. Wygodniej go jednak zapisać w postaci $3 + 5 \times x$. Jego graficzna reprezentacja jest przedstawiona na rysunku 8.1 na następnej stronie. Przykładem termu gatunku B jest $\leq(x, +(4, y))$. Możemy go czytelniej zapisać w postaci $x \leq 4 + y$.



Rysunek 8.1. Graficzne przedstawienie termu

Przykład 8.2. Rozważmy zbiór gatunków $\mathcal{S} = \{obiekt, fakt\}$, sygnaturę

$$\begin{aligned}
\Sigma^{obiekt} &= \{\text{pies, kot, stół}\} \\
\Sigma^{obiekt \to obiekt} &= \{\text{mały, wysoki, brzydki}\} \\
\Sigma^{obiekt \to fakt} &= \{\text{biegnie, śpi}\} \\
\Sigma^{obiekt \times obiekt \to fakt} &= \{\text{gryzie, liże}\}
\end{aligned}$$

przy czym zbiory Σ^{τ} dla pozostałych typów τ są puste, oraz zbiory zmiennych

$$\begin{aligned}
\mathcal{X}^{obiekt} &= \{X, Y, Z, \ldots\} \\
\mathcal{X}^{fakt} &= \{P, Q, R, \ldots\}
\end{aligned}$$

Przykładami termów gatunku *obiekt* są

pies	mały(pies)	wysoki(brzydki(kot))
mały(mały(stół))	X	mały(X)

zaś gatunku *fakt* są

biegnie(wysoki(stół))	śpi(brzydki(kot))	liże(mały(pies), kot)
gryzie(X, brzydki(X))	Q	liże(X, Y)

Termy zdefiniowane wyżej nazywamy *termami pierwszego rzędu*. Za ich pomocą nie można wyrazić *kwantyfikacji* (*wiązania*) zmiennych. W teorii języków programowania wielkie znaczenie mają termy *wyższych rzędów*, tzw. *lambda termy* lub *lambda wyrażenia*, wyposażone w mechanizm wiązania zmiennych. Są one uniwersalnym językiem, w którym można wyrazić całą matematykę. Ich opis wykracza jednak znacznie poza ramy niniejszych skromnych notatek. Dodamy jedynie, że Standard ML-owa abstrakcja funkcyjna `fn` x `=>` E jest przykładem wyrażenia wyższego rzędu.

8.1.1. Termy nad pojedynczym gatunkiem

Bardzo często rozważa się szczególny przypadek, gdy zbiór gatunków $\mathcal{S}$ jest jednoelementowy. Wówczas typy różnią się jedynie arnością i nie potrzeba ich używać. Mamy wtedy

jeden zbiór zmiennych $\mathcal{X}$, zbiory Σ_n *symboli o arności n*, sygnaturę (zwaną sygnaturą *jednogatunkową*) $\Sigma = \{\Sigma_n\}_{n\geq 0}$ i jeden zbiór termów $\mathcal{T}(\Sigma, \mathcal{X})$, zadany następującą definicją indukcyjną:

1. $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$;
2. jeżeli $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X})$ dla $n \geq 0$ oraz $f \in \Sigma_n$, to $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})$;
3. każdy term można zbudować używając reguł 1 i 2.

Sygnaturę nad więcej niż jednoelementowym zbiorem gatunków będziemy nazywać sygnaturą *wielogatunkową*.

8.2. Unifikacja

8.2.1. Podstawienie

Nie wyjaśniliśmy do tej pory roli zmiennych w termach. *Zbiór zmiennych termu t*, oznaczany FV(t) (od ang. *free variables*; tu akurat wszystkie zmienne są wolne), definiujemy indukcyjnie:

$$\begin{aligned} \mathrm{FV}(x) &= \{x\}, \qquad \text{dla } x \in \mathcal{X} \\ \mathrm{FV}(f(t_1, \ldots, t_n)) &= \mathrm{FV}(t_1) \cup \ldots \cup \mathrm{FV}(t_n) \end{aligned}$$

Termy nie zawierające zmiennych nazywamy *termami stałymi* (ang. *ground terms*). Zatem term t jest stały, gdy $\mathrm{FV}(t) = \emptyset$. Zbiór termów stałych gatunku a to $\mathcal{T}^a(\Sigma, \emptyset)$. Termy stałe, np. mały(pies) z przykładu 8.2 nazywają ustalone obiekty. Zmienne w termach pozwalają opisać całe zbiory takich obiektów, np. mały(X) nie nazywa pojedynczego obiektu, tylko jest *schematem*, opisuje zbiór wszystkich termów mały(t), gdzie t jest dowolnym termem gatunku *obiekt*, np. mały(kot), mały(pies) itd. Term mały(t) otrzymujemy *podstawiając* term t w miejsce zmiennej X w termie mały(X). Formalnie *podstawienie* jest skończonym zbiorem par zmiennych i termów zapisywanym w postaci

$$\theta = [x_1/t_1, \ldots, x_n/t_n]$$

gdzie $x_i \in \mathcal{X}^{a_i}$ i $t_i \in \mathcal{T}^{a_i}(\Sigma, \mathcal{X})$, dla $i = 1, \ldots, n$. Zwróćmy uwagę, że gatunek zmiennej x_i musi się zgadzać z gatunkiem termu t_i. Wynik podstawienia $\theta = [x_1/t_1, \ldots, x_n/t_n]$ w termie t oznaczamy $t\theta$ (a więc w tzw. zapisie postfiksowym, por. podrozdział 3.2) i definiujemy indukcyjnie:

$$\begin{aligned} x_i\theta &= t_i, \quad \text{dla } i = 1, \ldots, n \\ y\theta &= y, \quad \text{dla } y \in \mathcal{X}, y \neq x_i \text{ dla } i = 1, \ldots, n \\ f(s_1, \ldots, s_m)\theta &= f(s_1\theta, \ldots, s_m\theta) \end{aligned}$$

Dla przykładu mały$(X)[X/\text{kot}] = $ mały(kot), zaś gryzie$(X, Y)[Y/X] = $ gryzie(X, X). Zatem na podstawienie można patrzeć jak na odwzorowanie

$$\theta : \bigcup_{a\in\mathcal{S}} \mathcal{T}^a(\Sigma, \mathcal{X}) \to \bigcup_{a\in\mathcal{S}} \mathcal{T}^a(\Sigma, \mathcal{X})$$

które termom przyporządkowuje termy (tego samego gatunku).

Fakt 8.3. Jeżeli $t \in \mathcal{T}^a(\Sigma, \mathcal{X})$ i θ jest podstawieniem, to $t\theta \in \mathcal{T}^a(\Sigma, \mathcal{X})$.

Podstawienia można *składać*, tak jak wszelkie odwzorowania. Formalnie *złożeniem podstawień* θ_1 i θ_2 nazywamy podstawienie zapisywane $\theta_1\theta_2$, takie że

$$t(\theta_1\theta_2) \quad = \quad (t\theta_1)\theta_2$$

dla każdego termu $t \in \bigcup_{a\in\mathcal{S}} \mathcal{T}^a(\Sigma, \mathcal{X})$.

Zbiór zmiennych $\mathrm{Dom}(\theta) = \{x_1, \ldots, x_n\}$ nazywamy *dziedziną* (*nośnikiem*) podstawienia $\theta = [x_1/t_1, \ldots, x_n/t_n]$. Podstawienie *identycznościowe* (o pustej dziedzinie) oznaczamy []. Dla każdego termu $t \in \bigcup_{a\in\mathcal{S}} \mathcal{T}^a(\Sigma, \mathcal{X})$ zachodzi $t[] = t$. Podstawienie [] jest zatem faktycznie identycznością. Dla podstawień

$$\begin{aligned} \theta_1 &= [x_1/t_1, \ldots, x_n/t_n] \\ \theta_2 &= [y_1/s_1, \ldots, y_m/s_m] \end{aligned}$$

o rozłącznych dziedzinach definiujemy ich *sumę* wzorem

$$\theta_1 \cup \theta_2 \quad = \quad [x_1/t_1, \ldots, x_n/t_n, y_1/s_1, \ldots, y_m/s_m]$$

Lemat 8.4. Jeżeli podstawienia $\theta_1 = [x_1/t_1, \ldots, x_n/t_n]$ i θ_2 mają rozłączne dziedziny, to

$$\theta_1\theta_2 \quad = \quad [x_1/t_1\theta_2, \ldots, x_n/t_n\theta_2] \cup \theta_2$$

Mówimy, że term s jest *ukonkretnieniem* (*konkretyzacją*, *instancją*) termu t, jeżeli istnieje podstawienie θ, takie, że $t\theta = s$.

Na podstawieniach wprowadzamy relację częściowego praporządku $\leq$: mówimy, że podstawienie θ_1 jest *co najmniej tak ogólne*, jak podstawienie θ_2, co zapisujemy $\theta_1 \leq \theta_2$, jeżeli istnieje podstawienie ρ, takie że $\theta_1\rho = \theta_2$. Niech $\theta_1 \sim \theta_2$ jeśli $\theta_1 \leq \theta_2$ i $\theta_2 \leq \theta_1$.

Fakt 8.5. Relacja $\leq$ na podstawieniach jest częściowym praporządkiem (jest zwrotna i przechodnia), zaś $\sim$ jest relacją równoważności. Relacja $\leq$ na klasach równoważności

$$[\theta_1]_\sim \leq [\theta_2]_\sim \quad \Longleftrightarrow \quad \theta_1 \leq \theta_2$$

jest poprawnie określona i jest częściowym porządkiem (jest zwrotna, przechodnia i słabo antysymetryczna). Od tej pory będziemy często utożsamiać podstawienia równoważne i *de facto* rozważać nie podstawienia, tylko ich klasy abstrakcji modulo $\sim$. Każda para podstawień ma wówczas kres dolny, tj. *najmniej ogólne uogólnienie*, oznaczane $\theta_1 \wedge \theta_2$ (rodzina podstawień z relacją $\leq$ jest *dolną półkratą*). Elementem najmniejszym jest podstawienie identycznościowe []. Kres górny pary podstawień nie zawsze istnieje, elementu największego w zbiorze podstawień nie ma (z wyjątkiem przypadków o zdegenerowanej sygnaturze).

$R \leftarrow \{t \stackrel{?}{=} s\}$, gdzie $t \stackrel{?}{=} s$ jest równaniem do rozwiązania
$\theta \leftarrow []$
dopóki $R \neq \emptyset$ wykonuj poniższe czynności:
wybierz dowolne równanie $t \stackrel{?}{=} s$ z R i usuń je z R
jeżeli $t \stackrel{?}{=} s$ jest postaci
$x \stackrel{?}{=} x$ dla $x \in \mathcal{X}$, to pomiń je
$x \stackrel{?}{=} t$ lub $t \stackrel{?}{=} x$, gdzie $x \in \mathcal{X}$ i $x \neq t$, to
jeśli $x \notin \mathrm{FV}(t)$, to
$R \leftarrow R[x/t]$
$\theta \leftarrow \theta[x/t]$
w przeciwnym razie koniec, unifikator nie istnieje
$f(t_1, \ldots, t_n) \stackrel{?}{=} f(s_1, \ldots, s_n)$, to
$R \leftarrow R \cup \{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\}$
$f(t_1, \ldots, t_n) \stackrel{?}{=} g(s_1, \ldots, s_m)$, przy czym $f \neq g$, to
koniec, unifikator nie istnieje
θ jest poszukiwanym najogólniejszym unifikatorem

Tablica 8.1. Algorytm znajdowania najbardziej ogólnego unifikatora pary termów

8.2.2. Algorytm unifikacji

Równaniem nazywamy parę termów tego samego gatunku. Równanie zapisujemy zwykle w postaci $t \stackrel{?}{=} s$. Zbiór równań nazywamy *układem równań*. Podstawienie θ, takie że $t\theta = s\theta$ nazywamy *unifikatorem* pary termów t i s. Ogólniej, unifikatorem układu równań $\{t_i \stackrel{?}{=} s_i\}_{i=1}^n$ nazywamy podstawienie θ, takie że $t_i\theta = s_i\theta$ dla $i = 1, \ldots, n$. Najbardziej ogólny unifikator termów t i s oznaczamy $\mathrm{mgu}(t, s)$ (ang. *most general unifier*). Najbardziej ogólny unifikator układu równań oznaczamy $\mathrm{mgu}\{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\}$. Najbardziej ogólny unifikator nie zawsze istnieje, jeśli jednak zbiór unifikatorów danego równania jest niepusty, to istnieje wśród nich unifikator najbardziej ogólny. *Zadanie unifikacji* to problem znalezienia najbardziej ogólnego unifikatora dla zadanego równania (lub ogólniej układu równań). Algorytm znajdowania najbardziej ogólnego unifikatora pary termów jest przedstawiony w tablicy 8.1. Zmienne występujące w tym algorytmie, to układ równań R i podstawienie θ. Początkowo R zawiera wejściowe równanie do rozwiązania, $R = \{s \stackrel{?}{=} t\}$, zaś θ jest podstawieniem identycznościowym, $\theta = []$. W głównej pętli algorytmu są wykonywane pewne transformacje pary $\langle R, \theta\rangle$. Napis $R[x/t]$ oznacza wynik podstawienia $[x/t]$ we wszystkich termach układu równań R, natomiast $\theta[x/t]$ jest złożeniem podstawień θ i $[x/t]$. Dowód poprawności algorytmu polega na uzasadnieniu następującego faktu.

Fakt 8.6. Niech $\langle R', \theta'\rangle$ będzie wynikiem wykonania dowolnej transformacji występującej w algorytmie unifikacji. Wówczas

$$\{\theta\rho \mid \rho \text{ jest unifikatorem } R\} \quad = \quad \{\theta'\rho' \mid \rho' \text{ jest unifikatorem } R'\}$$

Niech $s \stackrel{?}{=} t$ będzie wejściowym równaniem, θ zaś wynikiem pracy algorytmu (za chwilę uzasadnimy, że algorytm zawsze się zatrzymuje). Z faktu 8.6 wynika natychmiast przez indukcję względem liczby kroków algorytmu, że

$$\{\rho \mid \rho \text{ jest unifikatorem równania } s \stackrel{?}{=} t\} \quad = \quad \{\theta\rho \mid \text{dla dowolnego } \rho\}$$

gdyż na końcu układ R jest pusty, każde podstawienie ρ jest więc jego unifikatorem. Zatem wszystkie unifikatory równania $s \stackrel{?}{=} t$ są postaci $\theta\rho$. Najogólniejszym z nich jest oczywiście θ. W przypadkach, w których algorytm twierdzi, że unifikator nie istnieje łatwo sprawdzić, że nie istnieje unifikator wyróżnionego w danym kroku równania, a więc i całego układu R. Na mocy faktu 8.6 nie istnieje zatem również unifikator wyjściowego równania. Aby dowieść, że algorytm się nie zapętla, definiujemy pewną „miarę złożoności" układu R. Jest to para nieujemnych liczb całkowitych $\langle n, m\rangle$, gdzie n jest liczbą zmiennych występujących w R, zaś m — liczbą wystąpień wszystkich zmiennych i symboli w R. Zbiór par nieujemnych liczb całkowitych jest dobrze uporządkowany relacją porządku leksykograficznego. Pozostaje sprawdzić, że wykonanie jakiejkolwiek transformacji układu $\langle R, \theta\rangle$ powoduje zmniejszenie naszej miary złożoności. Ponieważ w zbiorze dobrze uporządkowanym nie istnieją nieskończone ciągi ściśle malejące, po wykonaniu skończonej liczby kroków algorytm musi się zatrzymać.

Przykład 8.7. Niech sygnatura jednogatunkowa Σ zawiera binarny symbol f i unarny symbol g. Zmiennymi są X, Y, Z itd. Znajdziemy najogólniejszy unifikator równania

$$f(X,\ f(Y, g(Y))) \stackrel{?}{=} f(Z, Z)$$

Początkowo $R = \{f(X,\ f(Y, g(Y))) \stackrel{?}{=} f(Z, Z)\}$, wybieramy zatem wejściowe równanie. Pasuje do niego przedostatnia tranformacja opisana w algorytmie unifikacji. Tworzymy zatem nowy układ $R = \{X \stackrel{?}{=} Z,\ f(Y, g(Y)) \stackrel{?}{=} Z\}$, a podstawienie θ nadal jest identycznościowe. Teraz możemy wybrać jedno z dwóch równań. Rozważmy np. pierwsze. Jest ono postaci $x \stackrel{?}{=} t$, gdzie $x \notin \mathrm{FV}(t)$. Zatem w drugim równaniu układu R dokonujemy podstawienia $[X/Z]$ i podstawienie to składamy z podstawieniem θ. Otrzymujemy $R = \{f(Y, g(Y)) \stackrel{?}{=} Z\}$ i $\theta = [X/Z]$. W kolejnym kroku algorytmu rozważamy równanie $f(Y, g(Y)) \stackrel{?}{=} Z$. Podobnie jak w poprzednim, jest ono postaci $x \stackrel{?}{=} t$, gdzie $x \notin \mathrm{FV}(t)$. Mamy więc $R = \emptyset$ i $\theta = [X/Z][Z/f(Y, g(Y))] = [X/f(Y, g(Y)), Z/f(Y, g(Y))]$, gdzie ostatnia równość jest prawdziwa na mocy lematu 8.4. Najogólniejszym unifikatorem równania $f(X,\ f(Y, g(Y))) \stackrel{?}{=} f(Z, Z)$ jest zatem

$$[X/f(Y, g(Y)), Z/f(Y, g(Y))]$$

Zauważmy, że proces rozwiązywania układu równań R bardzo przypomina metodę rozwiązywania układów równań liniowych zwaną *eliminacją Gaussa*. *Eliminacji struktury* (transformacji równania $f(t_1, \ldots, t_n) \stackrel{?}{=} f(s_1, \ldots, s_n)$ do układu $\{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\}$) odpowiada *normalizacja* równania liniowego, *eliminacji zmiennej* (usunięciu równania $x \stackrel{?}{=} t$ z układu) — podobna eliminacja w algorytmie Gaussa. Tam również w miejsce eliminowanej

$$\frac{}{t=t}\ \text{(Refl)} \qquad \frac{t=s}{s=t}\ \text{(Sym)} \qquad \frac{s=r \quad r=t}{s=t}\ \text{(Trans)} \qquad \frac{t=s \quad r=u}{t[x/r]=s[x/u]}\ \text{(Mon)}$$

Tablica 8.2. Reguły wnioskowania dla równościowych teorii syntaktycznych

zmiennej wstawia się jej wyliczoną wartość. W algorytmie Gaussa otrzymujemy ostatecznie układ w tzw. *postaci rozwikłanej*. W algorytmie unifikacji rolę takiego układu pełni podstawienie.

8.3. Teorie syntaktyczne

Termy są narzędziem pozwalającym nazywać pewne obiekty. Dotychczas żąglowaliśmy tymi nazwami nie próbując nadać im żadnego znaczenia. Sygnatura, określająca postać termów, mówi jedynie, co jest poprawną wypowiedzią, a co nie. Gra rolę słownika ortograficznego. Trudno byłoby zrozumieć sens zdania w nieznanym nam języku posługując się przy tym słownikiem ortograficznym! Słowniki wyjaśniające znaczenie słów podają zwykle równoważne opisy tej samej rzeczy na zasadzie „*to* jest to to samo co *tamto*". Możemy podobnie postąpić z termami, definiując pojęcie *równości*. Jest to para termów tego samego gatunku, zawierających przeważnie zmienne, zapisana w postaci $t = s$, np. $1 + 2 = 3$, $(x+y)\times z = x\times z + y\times z$ lub prawdziwy(przyjaciel) = Cudak. Wybrane równości przyjmujemy za spełnione *ad hoc* i nazywamy *aksjomatami równościowymi*. Wyliczając aksjomaty pragniemy podać zasady utożsamiania termów, uważania ich za równoważne. Definiujemy zatem pewną relację równości termów. Relacja równości powinna być relacją równoważności: być zwrotna, przechodnia i symetryczna. Nadto powinna być *monotoniczna*: jeżeli uznaliśmy termy t i s (być może zawierające zmienną x) za równe, to cokolwiek podstawilibyśmy za zmienną x w obu termach jednocześnie nie powinno tej relacji równości zaburzyć. Np. jeśli $x + y = y + x$, to także $(z + w) + y = y + (z + w)$, $1 + 2 = 2 + 1$ itd. Za zmienną x w obu termach nie potrzeba nawet wstawiać tego samego termu. Wystarczy, że wstawimy tam termy, o których wiemy, że są równe. Zatem będziemy mówić, że binarna, określona na termach relacja R jest *monotoniczna*, jeżeli

$$t\,Rs \wedge r\,Ru \implies (t[x/r])R(s[x/u])$$

Dla uproszczenia definicji rozważmy zbiór termów $\mathcal{T}(\Sigma, \mathcal{X})$ nad jednogatunkową sygnaturą Σ i zbiorem zmiennych $\mathcal{X}$ (przypadek sygnatury wielogatunkowej można rozważyć analogicznie, definicje się jednak nieco komplikują). *Równościową teorią syntaktyczną* zadaną przez zbiór aksjomatów A i oznaczaną $\mathrm{Th}^{\vdash}(A)$ nazywamy najmniejszą monotoniczną relację równoważności zawierającą zbiór A. Niekiedy będziemy pisać $A \vdash t = s$ na oznaczenie faktu, że $(t = s) \in \mathrm{Th}^{\vdash}(A)$ i mówić, że równość $t = s$ jest *twierdzeniem teorii* $\mathrm{Th}^{\vdash}(A)$. Poprawność definicji wymaga dowodu (elementy najmniejsze nie zawsze istnieją). Niech $\{R_\kappa\}_\kappa$ będzie rodziną wszystkich monotonicznych relacji równoważności zawierających zbiór A. Rodzina ta jest niepusta, bo należy do niej relacja totalna (zawierająca wszyst-

kie pary termów). Ponieważ przekrój dowolnej liczby monotonicznych relacji równoważności jest monotoniczną relacją równoważności, to $\bigcap_\kappa R_\kappa$ jest najmniejszą monotoniczną relacją równoważności zawierającą zbiór A. Definicja jest zatem poprawna.

Relację $\mathrm{Th}^{\vdash}(A)$ można też budować indukcyjnie. Niech

$$\begin{aligned} R_0 &= A \cup \{t = t \mid t \in \mathcal{T}(\Sigma, \mathcal{X})\} \\ R_{n+1} &= R_n \cup \{s = t \mid (t = s) \in R_n\} \cup \{s = t \mid (s = u), (u = t) \in R_n\} \cup \\ &\quad \{t[x/r] = s[x/u] \mid (t = s), (r = u) \in R_n\} \\ R &= \bigcup_{n=0}^{\infty} R_n \end{aligned}$$

Wówczas $R = \mathrm{Th}^{\vdash}(A)$. Wpierw pokazujemy indukcyjnie względem n, że $R_n \subseteq \mathrm{Th}^{\vdash}(A)$ dla każdego $n \geq 0$, więc $R \subseteq \mathrm{Th}^{\vdash}(A)$. Następnie zauważamy, że R jest monotoniczną relacją równoważności zawierającą A, mamy więc $\mathrm{Th}^{\vdash}(A) \subseteq R$.

Aby sprawdzić, czy równość $t = s$ należy do teorii $\mathrm{Th}^{\vdash}(A)$ możemy zacząć od aksjomatów ze zbioru A i równości $t = t$ dla $t \in \mathcal{T}(\Sigma, \mathcal{X})$ i stosować następujące reguły wnioskowania:

1. jeżeli $(t = s) \in \mathrm{Th}^{\vdash}(A)$, to także $(s = t) \in \mathrm{Th}^{\vdash}(A)$;
2. jeżeli $(s = r) \in \mathrm{Th}^{\vdash}(A)$ i $(r = t) \in \mathrm{Th}^{\vdash}(A)$, to także $(s = t) \in \mathrm{Th}^{\vdash}(A)$;
3. jeżeli $(t = s) \in \mathrm{Th}^{\vdash}(A)$ i $(r = u) \in \mathrm{Th}^{\vdash}(A)$, to także $(t[x/r] = s[x/u]) \in \mathrm{Th}^{\vdash}(A)$;

tak długo, aż dojdziemy do równości, która nas interesuje. Powyższy mechanizm można wygodnie opisać w postaci tzw. *formalnego systemu wnioskowania*, składającego się ze zbioru aksjomatów A i zestawu czterech *reguł wnioskowania* przedstawionych w tablicy 8.2 na poprzedniej stronie. Równość $t = s$ jest twierdzeniem teorii $\mathrm{Th}^{\vdash}(A)$, jeżeli istnieje *drzewo dowodu*, w którego korzeniu znajduje się równość $t = s$ a w liściach — aksjomaty ze zbioru A lub równość $t = t$, tj. aksjomat (Refl). Przykład drzewa dowodu jest przedstawiony w tablicy 8.3 na następnej stronie.

Przykład 8.8 (Półgrupy przemienne z jednością). Rozważmy jednogatunkową sygnaturę złożoną ze stałej e i binarnego symbolu $\oplus$. Niech $\mathcal{X} = \{x, y, z \ldots\}$ będzie zbiorem zmiennych. Teoria przemiennych półgrup z jednością jest zadana przez następujący zbiór równości:

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{8.1}$$
$$e \oplus x = x \tag{8.2}$$
$$x \oplus y = y \oplus x \tag{8.3}$$

Dla przykładu potrafimy pokazać, że $x \oplus e = x$ jest twierdzeniem tej teorii. Dowód jest przedstawiony w tablicy 8.3 na następnej stronie.

$$\dfrac{\dfrac{\dfrac{}{x \oplus y = y \oplus x}\,(8.3)}{x \oplus e = e \oplus x}\,(\mathrm{Mon}) \qquad \dfrac{}{e \oplus x = x}\,(8.2)}{x \oplus e = x}\,(\mathrm{Trans})$$

Tablica 8.3. Dowód twierdzenia w teorii półgrup przemiennych

Przykład 8.9 (Rachunek kombinatorów). Rozważmy sygnaturę jednogatunkową złożoną z dwóch stałych K i S i binarnego infiksowego symbolu $\cdot$. Umawiamy się, że podobnie jak symbol mnożenia w arytmetyce, symbol $\cdot$ będziemy często opuszczać, pisząc np. KS zamiast $\mathsf{K} \cdot \mathsf{S}$. Niech $\mathcal{X} = \{x, y, z \ldots\}$ będzie zbiorem zmiennych. Termy nad powyższą sygnaturą będziemy nazywać *kombinatorami*. Teorię równościową zadaną przez aksjomaty

$$\begin{aligned} \mathsf{K}xy &= x \\ \mathsf{S}xyz &= xz(yz) \end{aligned}$$

nazywamy *rachunkiem kombinatorów*. Jeżeli popatrzymy na kombinatory jak na obiekty reprezentujące funkcje, a na operator $\cdot$ jak na operację aplikacji funkcji do argumentu (wówczas np. K jest *projekcją na pierwszą współrzędną*, tj. funkcją dwu zmiennych zwracającą swój pierwszy argument), to okaże się, że w tak prostej teorii można opisać wszystkie funkcje obliczalne. Dokładniejsze zbadanie tej teorii wykracza jednak znacznie poza ramy naszych bieżących rozważań. Podamy zatem tylko dwa przykłady: $I = \mathsf{SKK}$ oraz $K' = \mathsf{SK}$. Zauważmy, że I reprezentuje funkcję identyczności, bowiem $Ix = \mathsf{SKK}x = \mathsf{K}x(\mathsf{KK}) = x$, zaś K' jest projekcją na drugą współrzędną, tj. $K'xy = y$, bo $K'xy = \mathsf{SK}xy = \mathsf{K}y(xy) = y$.

8.4. Algebry

Niech $\Sigma = \{\Sigma^a\}_{a \in \mathcal{S}}$ będzie sygnaturą nad zbiorem gatunków $\mathcal{S}$. *Algebrą* (*strukturą algebraiczną*) o sygnaturze Σ nazywamy parę $\mathfrak{A} = \langle \mathcal{A}, \cdot^{\mathfrak{A}} \rangle$ złożoną z rodziny $\mathcal{A} = \{A^a\}_{a \in \mathcal{S}}$ zbiorów zwanych *dziedzinami* (*nośnikami*, *uniwersami*) algebry (po jednym dla każdego gatunku) i odwzorowania $\cdot^{\mathfrak{A}}$ zwanego *interpretacją symboli funkcyjnych*, które każdemu symbolowi sygnatury $f \in \Sigma^{a_1 \times \ldots \times a_n \to b}$ przyporządkowuje funkcję

$$f^{\mathfrak{A}} : A^{a_1} \times \ldots \times A^{a_n} \to A^b$$

Przykład 8.10. Rozważmy sygnaturę jednogatunkową $\Sigma = \{\Sigma_n\}_{n=0}^{\infty}$, w której $\Sigma_0 = \{e\}$, $\Sigma_2 = \{\otimes\}$ i pozostałe zbiory symboli są puste. Przykładami algebr o sygnaturze Σ są

1. addytywna półgrupa liczb naturalnych $\mathfrak{N}^+ = \langle \mathbb{N}, \cdot^{\mathfrak{N}} \rangle$, w której $\mathbb{N}$ jest zbiorem liczb naturalnych, $e^{\mathfrak{N}^+} = 0$, a $\otimes^{\mathfrak{N}^+}$ jest operacją dodawania;
2. multiplikatywna półgrupa liczb naturalnych $\mathfrak{N}^\times = \langle \mathbb{N}, \cdot^{\mathfrak{N}^\times} \rangle$, w której $\mathbb{N}$ jest zbiorem liczb naturalnych, $e^{\mathfrak{N}^\times} = 1$, a $\otimes^{\mathfrak{N}^\times}$ jest operacją mnożenia;

3. półgrupa słów $\mathfrak{W} = \langle \Xi^*, \cdot^{\mathfrak{W}} \rangle$, w której Ξ^* jest zbiorem słów nad pewnym alfabetem Ξ, $e^{\mathfrak{W}}$ jest słowem pustym, a $\otimes^{\mathfrak{W}}$ jest operacją konkatenacji słów.

Przykład 8.11. Rozważmy zbiór gatunków $\mathcal{S} = \{B, N\}$ i sygnaturę, w której $\Sigma^B = \{\mathtt{T}, \mathtt{F}\}$, $\Sigma^N = \{0, 1, \ldots\}$, $\Sigma^{N\times N\to N} = \{+, *\}$, $\Sigma^{B\times B\to B} = \{|, \&\}$, $\Sigma^{B\times N\times N\to N} = \{?\}$, a pozostałe zbiory symboli są puste. Przykładem algebry o tej sygnaturze jest $\mathfrak{A} = \langle \{A^B, A^N\}, \cdot^{\mathfrak{A}} \rangle$, w której $A^B = \{T, F\}$, $A^N = \mathbb{N}$ (zbiór liczb naturalnych), $\mathtt{T}^{\mathfrak{A}} = T$, $\mathtt{F}^{\mathfrak{A}} = F$, $0^{\mathfrak{A}} = 0$, $1^{\mathfrak{A}} = 1$ itd, $+^{\mathfrak{A}}$ jest operacją dodawania liczb naturalnych, $*^{\mathfrak{A}}$ jest operacją mnożenia liczb naturalnych, $|^{\mathfrak{A}}$ jest alternatywą a $\&^{\mathfrak{A}}$ koniunkcją wartości logicznych, zaś $?^{\mathfrak{A}}(T, n, m) = n$ i $?^{\mathfrak{A}}(F, n, m) = m$ dla dowolnych liczb naturalnych $n, m \in \mathbb{N}$.

Przykład 8.12. Niech Σ będzie sygnaturą nad zbiorem gatunków $\mathcal{S}$ a $\mathcal{X}$ rodziną zmiennych oraz

$$f^{\mathfrak{T}}(t_1, \ldots, t_n) \quad = \quad f(t_1, \ldots, t_n)$$

dla $f \in \Sigma^{a_1\times\ldots\times a_n\to b}$, $t_i \in \mathcal{T}^{a_i}(\Sigma, \mathcal{X})$ dla $i = 1, \ldots, n$ i $a_1, \ldots, a_n, b \in \mathcal{S}$. Wówczas $\mathfrak{T}(\Sigma, \mathcal{X}) = \langle \{T^a(\Sigma, \mathcal{X})\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{T}} \rangle$ jest algebrą o sygnaturze Σ. Nazywamy ją *algebrą termów*.

Rozważmy algebrę $\mathfrak{A} = \langle \{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}} \rangle$ o sygnaturze Σ nad zbiorem gatunków $\mathcal{S}$. Niech $\mathcal{X} = \{\mathcal{X}^a\}_{a\in\mathcal{S}}$ będzie rodziną zmiennych. Dowolną rodzinę odwzorowań $\eta = \{\eta^a\}_{a\in\mathcal{S}}$, takich że $\eta^a : \mathcal{X}^a \to A^a$ nazywamy *interpretacją zmiennych* w algebrze $\mathfrak{A}$. Interpretacja symboli funkcyjnych wraz z interpretacją zmiennych jednoznacznie zadają interpretację $[\![\cdot]\!]^{\mathfrak{A}}_\eta$ dowolnych termów w algebrze $\mathfrak{A}$:

$$\begin{aligned} [\![x^a]\!]^{\mathfrak{A}}_\eta &= \eta^a(x^a) \\ [\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_\eta &= f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_\eta, \ldots, [\![t_n]\!]^{\mathfrak{A}}_\eta) \end{aligned}$$

dla $x^a \in \mathcal{X}^a$, $f \in \Sigma^{a_1\times\ldots\times a_n\to b}$ i $t_i \in \mathcal{T}^{a_i}(\Sigma, \mathcal{X})$, $i = 1, \ldots, n$ oraz $a, a_1, \ldots, a_n, b \in \mathcal{S}$.

Fakt 8.13. Jeżeli wartościowania zmiennych η_1 i η_2 zgadzają się na zbiorze zmiennych termu t, tj. $\eta_1^a(x^a) = \eta_2^a(x^a)$ dla $x^a \in \mathrm{FV}(t)$, to $[\![t]\!]^{\mathfrak{A}}_{\eta_1} = [\![t]\!]^{\mathfrak{A}}_{\eta_2}$. W szczególności wartościowanie termu stałego nie zależy od wartościowania zmiennych. Będziemy więc pisać $[\![t]\!]^{\mathfrak{A}}$, pomijając wartościowanie zmiennych, gdy $\mathrm{FV}(t) = \emptyset$.

8.4.1. Homomorfizmy algebr

Rozważmy dwie algebry $\mathfrak{A} = \langle \{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}} \rangle$ i $\mathfrak{B} = \langle \{B^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{B}} \rangle$ o tej samej sygnaturze Σ. Rodzinę odwzorowań $h = \{h^a\}_{a\in\mathcal{S}}$, gdzie $h^a : A^a \to B^a$ dla $a \in \mathcal{S}$, nazywamy *homomorfizmem*, jeżeli

$$h^b(f^{\mathfrak{A}}(u_1, \ldots, u_n)) \quad = \quad f^{\mathfrak{B}}(h^{a_1}(u_1), \ldots, h^{a_n}(u_n))$$

dla $f \in \Sigma^{a_1\times\ldots\times a_n\to b}$, $u_i \in A_i$ dla $i = 1, \ldots, n$ oraz $a_1, \ldots, a_n, b \in \mathcal{S}$. Homomorfizm będący bijekcją (odwzorowaniem różnowartościowym i „na") nazywamy *izomorfizmem*. Jeżeli

istnieje izomorfizm z algebry $\mathfrak{A}$ na algebrę $\mathfrak{B}$, to mówimy, że algebry $\mathfrak{A}$ i $\mathfrak{B}$ są *izomorficzne*. Izomorfizm algebr jest relacją równoważności.

Przykład 8.14. Rozważmy algebry $\mathfrak{R}^+ = \langle \mathbb{R}, \cdot^{\mathfrak{R}^+} \rangle$ i $\mathfrak{R}^\times = \langle \mathbb{R}^+, \cdot^{\mathfrak{R}^\times} \rangle$ nad jednogatunkową sygnaturą zawierającą stałą e i binarny symbol $\otimes$, gdzie $\mathbb{R}$ jest zbiorem liczb rzeczywistych a $\mathbb{R}^+$ zbiorem liczb rzeczywistych dodatnich, $\otimes^{\mathfrak{R}^+}$ jest dodawaniem a $\otimes^{\mathfrak{R}^\times}$ mnożeniem liczb rzeczywistych, $e^{\mathfrak{R}^+} = 0$ i $e^{\mathfrak{R}^\times} = 1$. Wówczas funkcja logarytmiczna $\ln : \mathbb{R}^+ \to \mathbb{R}$ jest homomorfizmem z algebry $\mathfrak{R}^\times$ w algebrę $\mathfrak{R}^+$, bo

$$\ln\left(e^{\mathfrak{R}^\times}\right) = \ln(1) = 0 = e^{\mathfrak{R}^+}$$
$$\ln\left(u_1 \otimes^{\mathfrak{R}^\times} u_2\right) = \ln(u_1 \times u_2) = \ln(u_1) + \ln(u_2) = \ln(u_1) \otimes^{\mathfrak{R}^+} \ln(u_2)$$

dla dowolnych dodatnich liczb rzeczywistych $u_1, u_2 \in \mathbb{R}^+$. Ponieważ jest ona także bijekcją, to jest izomorfizmem algebr $\mathfrak{R}^\times$ i $\mathfrak{R}^+$.

Przykład 8.15. W każdej algebrze $\mathfrak{A}$ wartościowanie termów dla dowolnego wartościowania zmiennych jest homomorfizmem z algebry termów w algebrę $\mathfrak{A}$.

Przykład 8.16. Podstawienie jest homomorfizmem z algebry termów w nią samą.

8.4.2. Podalgebry i algebry generowane

Podalgebrą algebry $\mathfrak{A} = \langle \{A^a\}_{a \in \mathcal{S}}, \cdot^{\mathfrak{A}} \rangle$ o sygnaturze Σ nad zbiorem gatunków $\mathcal{S}$ nazywamy rodzinę zbiorów $\{B^a\}_{a \in \mathcal{S}}$, taką, że:

1. $B^a \subseteq A^a$ dla $a \in \mathcal{S}$;
2. $f^{\mathfrak{A}}(u_1, \ldots, u_n) \in B^b$ dla $f \in \Sigma^{a_1 \times \ldots \times a_n \to b}$, $u_i \in B^{a_i}$, $a_1, \ldots, a_n, b \in \mathcal{S}$ (zbiory B^a są zamknięte ze względu na działania $f^{\mathfrak{A}}$).

Podalgebra $\{B^a\}_{a \in \mathcal{S}}$ jest algebrą $\mathfrak{B} = \langle \{B^a\}_{a \in \mathcal{S}}, \cdot^{\mathfrak{B}} \rangle$ o sygnaturze Σ, jeśli położyć $f^{\mathfrak{B}} = f^{\mathfrak{A}} \restriction_{B^{a_1} \times \ldots \times B^{a_n}}$ dla $f \in \Pi^{a_1 \times \ldots \times a_n \to b}$ i $a_1, \ldots, a_n, b \in \mathcal{S}$ (działania $f^{\mathfrak{B}}$ są obcięciami działań $f^{\mathfrak{A}}$ do dziedzin algebry $\mathfrak{B}$).

Niech $\mathcal{G} = \{G^a\}_{a \in \mathcal{S}}$ będzie rodziną zbiorów, taką, że $G^a \subseteq A^a$. *Algebrą generowaną* przez rodzinę $\mathcal{G}$ nazywamy najmniejszą (w sensie relacji inkluzji dziedzin) podalgebrę $\mathfrak{A}(\mathcal{G}) = \langle \{B^a\}_{a \in \mathcal{S}}, \cdot^{\mathfrak{A}(\mathcal{G})} \rangle$ algebry $\mathfrak{A}$, taką, że $G^a \subseteq B^a$ dla każdego $a \in \mathcal{S}$. Uzasadnienia wymaga poprawność powyższej definicji (postulujemy, by odpowiednie zbiory były najmniejsze w pewnej klasie, podczas gdy elementy najmniejsze nie zawsze istnieją). Niech zatem $\{\mathfrak{B}_\kappa\}_\kappa$, gdzie $\mathfrak{B}_\kappa = \langle \{B^a_\kappa\}_{a \in \mathcal{S}}, \cdot^{\mathfrak{B}_\kappa} \rangle$, będzie rodziną wszystkich podalgebr algebry $\mathfrak{A}$, których dziedziny zawierają zbiory G^a. Rodzina ta jest niepusta, bo należy do niej sama algebra $\mathfrak{A}$. Niech $B^a = \bigcap_\kappa B^a_\kappa$ dla $a \in \mathcal{S}$. Rodzina $\{B^a\}_{a \in \mathcal{S}}$ jest, jak łatwo sprawdzić, podalgebrą algebry $\mathfrak{A}$. Nadto jest ona najmniejszą podalgebrą, której dziedziny zawierają zbiory G^a.

8.4.3. Zasada indukcji

Twierdzenie 8.17 (Zasada indukcji strukturalnej). Rozważmy rodzinę predykatów $\Phi = \{\Phi^a\}_{a\in\mathcal{S}}$ określonych na dziedzinach algebry $\mathfrak{A} = \langle\{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}}\rangle$ o sygnaturze Σ, tj. niech $\Phi^a \subseteq A^a$ dla $a \in \mathcal{S}$. Niech $\{G^a\}_{a\in\mathcal{S}}$ będzie rodziną zbiorów, taką, że $G^a \subseteq A^a$ dla $a \in \mathcal{S}$ i niech $\mathfrak{A}(\mathcal{G}) = \langle\{B^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}(\mathcal{G})}\rangle$ będzie algebrą generowaną przez rodzinę $\{G^a\}_{a\in\mathcal{S}}$. Jeżeli

1. $G^a \subseteq \Phi^a$ dla $a \in \mathcal{S}$;
2. $\Phi^{a_1}(u_1) \wedge \ldots \wedge \Phi^{a_n}(u_n) \Longrightarrow \Phi^b(f^{\mathfrak{A}}(u_1, \ldots, u_n))$ dla $f \in \Pi^{a_1\times\ldots\times a_n\to b}$, $u_i \in B^{a_i}$ i $a_1, \ldots, a_n, b \in \mathcal{S}$,

to $\Phi^a(u)$ dla każdego $u \in B^a$ i $a \in \mathcal{S}$.

Dowód. Z warunku 2 rodzina $\{\Phi^a\}_{a\in\mathcal{S}}$ jest podalgebrą algebry $\mathfrak{A}$. Z warunku 1 jest więc podalgebrą algebry $\mathfrak{A}$, której dziedziny zawierają zbiory G^a. Ponieważ algebra generowana jest najmniejszą algebrą o powyższych własnościach, więc $B^a \subseteq \Phi^a$ dla $a \in \mathcal{S}$.

Niech Σ będzie sygnaturą nad zbiorem gatunków $\mathcal{S}$, a $\mathfrak{T}$ algebrą termów stałych o sygnaturze Σ. Wtedy $\mathfrak{T}(\emptyset) = \mathfrak{T}$. Mamy więc w szczególności:

Twierdzenie 8.18 (Zasada indukcji strukturalnej dla termów). Niech Σ będzie sygnaturą nad zbiorem gatunków $\mathcal{S}$. Rozważmy rodzinę $\Phi = \{\Phi^a\}_{a\in\mathcal{S}}$ predykatów określonych na zbiorach termów $\{\mathcal{T}^a(\Sigma, \emptyset)\}_{a\in\mathcal{S}}$ nad sygnaturą Σ, tj. niech $\Phi^a \subseteq \mathcal{T}^a(\Sigma, \emptyset)$ dla $a \in \mathcal{S}$. Jeżeli

$$\Phi^{a_1}(t_1) \wedge \ldots \wedge \Phi^{a_n}(t_n) \quad\Longrightarrow\quad \Phi^b(f(t_1, \ldots, t_n))$$

dla $f \in \Sigma^{a_1\times\ldots\times a_n\to b}$, $t_i \in \mathcal{T}^{a_i}(\Sigma, \emptyset)$ i $a_1, \ldots, a_n, b \in \mathcal{S}$, to $\Phi^a(t)$ dla $t \in \mathcal{T}^a(\Sigma, \emptyset)$ i $a \in \mathcal{S}$.

8.4.4. Konstruktory

Niech $\mathfrak{A} = \langle\{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}}\rangle$ będzie algebrą o sygnaturze Σ i niech Γ będzie sygnaturą zawierającą wybrane symbole sygnatury Σ (tj. $\Gamma^\tau \subseteq \Sigma^\tau$ dla $\tau \in \mathbb{T}_1(\mathcal{S})$). Niech $\mathfrak{A}{\restriction_\Gamma} = \langle\{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}}{\restriction_\Gamma}\rangle$ będzie obcięciem algebry $\mathfrak{A}$ do sygnatury Γ. Jeżeli wartościowanie termów $[\![\cdot]\!]$ w algebrze $\mathfrak{A}$ jest izomorfizmem z algebry termów stałych $\mathfrak{T} = \langle\{T^a(\Gamma, \emptyset)\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{T}}\rangle$ o sygnaturze Γ na algebrę $\mathfrak{A}{\restriction_\Gamma}$, to sygnaturę Γ nazywamy *zbiorem konstruktorów* algebry $\mathfrak{A}$. Jeżeli Γ jest zbiorem konstruktorów algebry $\mathfrak{A} = \langle\{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}}\rangle$, to $\mathfrak{T} = \langle\{T^a(\Gamma, \emptyset)\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{T}}\rangle$ można rozważać jako algebrę o sygnaturze Σ, kładąc $f^{\mathfrak{T}}$ takie, by

$$[\![f^{\mathfrak{T}}(t_1, \ldots, t_n)]\!] \quad = \quad f^{\mathfrak{A}}([\![t_1]\!], \ldots, [\![t_n]\!])$$

dla f nie należących do Γ. Wówczas algebra termów $\mathfrak{T}$ jest izomorficzna z algebrą $\mathfrak{A}$.

Jeżeli Γ jest zbiorem konstruktorów algebry $\mathfrak{A}$, to każdy element algebry $\mathfrak{A}$ ma jednoznaczną nazwę w postaci termu nad sygnaturą Γ. Możemy także dowodzić własności algebry $\mathfrak{A}$ przez indukcję strukturalną względem zbioru konstruktorów Γ.

Nie każda algebra posiada zbiór konstruktorów. Niektóre zaś posiadają wiele zbiorów konstruktorów. Pojęcie zbioru konstruktorów ma wielkie znaczenie w teorii specyfikacji algebraicznych (por. rozdział 12).

8.5. Teorie semantyczne

Rozważmy algebrę $\mathfrak{A} = \langle \{A^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{A}} \rangle$ nad sygnaturą Σ. Powiemy, że równość $t = s$ jest *spełniona* w algebrze $\mathfrak{A}$, co oznaczamy $\mathfrak{A} \models t = s$, jeżeli $[\![t]\!]^{\mathfrak{A}}_{\eta} = [\![s]\!]^{\mathfrak{A}}_{\eta}$ dla każdego wartościowania zmiennych η. Jeżeli E jest zbiorem równości, to mówimy, że jest on *spełniony* w algebrze $\mathfrak{A}$, co oznaczamy $\mathfrak{A} \models E$, jeżeli $\mathfrak{A} \models t = s$ dla każdej równości $(t = s) \in E$. Zbiór równości spełnionych w algebrze $\mathfrak{A}$ oznaczamy $\mathrm{Th}(\mathfrak{A}) = \{t = s \mid \mathfrak{A} \models t = s\}$ i nazywamy *teorią algebry* $\mathfrak{A}$. Klasą algebr *zdefiniowaną* przez zbiór równości (aksjomatów) E nazywamy klasę $\mathcal{E}(E) = \{\mathfrak{A} \mid \mathfrak{A} \models E\}$. *Równościową teorią semantyczną* zadaną przez zbiór równości E nazywamy zbiór równości spełnionych w każdej algebrze spełniającej E:

$$\mathrm{Th}^{\models}(E) = \{(t = s) \mid \forall \mathfrak{A}.(\mathfrak{A} \models E \Rightarrow \mathfrak{A} \models t = s)\} = \bigcap_{\mathfrak{A}:\mathfrak{A}\models E} \mathrm{Th}(\mathfrak{A})$$

Twierdzenie 8.19. Dla każdego zbioru równości E jest $\mathrm{Th}^{\vdash}(E) = \mathrm{Th}^{\models}(E)$.

Możemy więc mówić po prostu o teorii równościowej $\mathrm{Th}(E)$ nie zaznaczając, czy jest ona syntaktyczna, czy semantyczna. Pojęcia te są bowiem zbieżne.

Można zadać pytanie, czy istnieje jedna konkretna algebra $\mathfrak{A}$, taka, że jej teoria jest teorią zadaną przez ustalony zbiór równości E, tj. czy istnieje algebra $\mathfrak{A}$, taka, że $\mathrm{Th}(\mathfrak{A}) = \mathrm{Th}(E)$. Nie zawsze tak jest. Jeśli jednak każda algebra spełniająca E ma wszystkie nośniki niepuste, to taką algebrą jest $\mathfrak{A} = \langle \mathcal{T}(\Sigma, \mathcal{X})/_{\sim}, \cdot^{\mathfrak{A}} \rangle$ gdzie $t \sim s \iff (t = s) \in \mathrm{Th}(E)$ i $f^{\mathfrak{A}}([t_1]_{\sim}, \ldots, [t_n]_{\sim}) = [f(t_1, \ldots, t_n)]_{\sim}$.

Niech $\mathcal{A}$ będzie klasą algebr o sygnaturze Σ. Algebra $\mathfrak{A} \in \mathcal{A}$ jest *algebrą początkową w klasie* $\mathcal{A}$, jeżeli dla każdej algebry $\mathfrak{B} \in \mathcal{A}$ istnieje dokładnie jeden homomorfizm z algebry $\mathfrak{A}$ w algebrę $\mathfrak{B}$. *Algebrą początkową dla zbioru równości* E nazywamy algebrę początkową w klasie $\mathcal{E}(E)$.

Twierdzenie 8.20. Algebrą początkową dla zbioru równości E jest algebra

$$\mathfrak{P} = \langle \mathcal{T}(\Sigma, \emptyset)/_{\sim}, \cdot^{\mathfrak{P}} \rangle$$

gdzie $t \sim s \iff (t = s) \in \mathrm{Th}(E)$ i $f^{\mathfrak{P}}([t_1]_{\sim}, \ldots, [t_n]_{\sim}) = [f(t_1, \ldots, t_n)]_{\sim}$.

W algebrze początkowej są spełnione te równości między termami *stałymi*, które są dowodliwe ze zbioru równości E. W algebrze początkowej mogą być jest spełnione inne równości (pomiędzy termami zawierającymi zmienne), które nie są dowodliwe ze zbioru równości E.

8.6. Składnia abstrakcyjna

W rozdziale 3 rozważaliśmy najprostszą formalizację pojęcia *języka* jako zbioru *napisów*, tj. ciągów znaków. W zastosowaniach, w których nie interesuje nas notacja (sposób zapisu),

$$
\begin{aligned}
\Sigma_D^A &= \{0, 1, 2, \ldots\} \\
\Sigma_D^B &= \{\texttt{true}, \texttt{false}\} \\
\Sigma_D^C &= \{\texttt{skip}\} \\
\Sigma_D^I &= \{\texttt{X}, \texttt{Y}, \texttt{Z}, \ldots\} \\
\Sigma_D^{I\to A} &= \{i\} \\
\Sigma_D^{A\to A} &= \{\texttt{-}\} \\
\Sigma_D^{A\times A\to A} &= \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \\
\Sigma_D^{B\to B} &= \{\texttt{!}\} \\
\Sigma_D^{A\times A\to B} &= \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\
\Sigma_D^{B\times B\to B} &= \{\texttt{||}, \texttt{\&\&}\} \\
\Sigma_D^{I\to C} &= \{\texttt{read}\} \\
\Sigma_D^{A\to C} &= \{\texttt{write}\} \\
\Sigma_D^{I\times A\to C} &= \{\texttt{=}\} \\
\Sigma_D^{C\times C\to C} &= \{\texttt{;}\} \\
\Sigma_D^{B\times C\to C} &= \{\texttt{if}, \texttt{while}\} \\
\Sigma_D^{B\times C\times C\to C} &= \{\texttt{if}\}
\end{aligned}
$$

Tablica 8.4. Sygnatura algebraiczna Σ_D opisująca abstrakcyjną składnię języka D

tylko budowa zdań należących do pewnego języka, np. gdy opisujemy semantykę (znaczenie) tych zdań, ta formalizacja jest zbyt szczegółowa i przez to mało przydatna. Wiele programów może mieć to samo znaczenie i różnić się jedynie sposobem zapisu. Na przykład *stałą* 1 możemy zapisać w języku D z podrozdziału 3.11 za pomocą *literałów* `1` lub `01`, wyrażenie $2 + 3$ zaś jako `2+3` albo `(2+3)`. Lepiej wówczas reprezentować program w postaci pewnych abstrakcyjnych obiektów, niezależnych od konkretnych rozwiązań składniowych. Opis języka jako zbioru ciągów znaków będziemy od tej pory nazywać *składnią konkretną*. Opis budowy języka za pomocą pewnych abstrakcyjnych obiektów, niezależnych od przyjętej notacji będziemy nazywać *składnią abstrakcyjną*. Ze składnią abstrakcyjną zetknęliśmy się już w podrozdziale 3.2, gdy mówiliśmy o abstrakcyjnych drzewach rozbioru wyrażeń. W składni abstrakcyjnej będziemy rozważać np. *stałe* (reprezentowane przez *literały* w składni konkretnej), *nazwy komórek pamięci* (reprezentowane przez *identyfikatory* w składni konkretnej) itp. Składnia abstrakcyjna ma ścisły związek z semantyką języka. Nie zawiera natomiast informacji o sposobie zapisu programu, np. o priorytetach operatorów i ich kierunkach łączności, o rozstawieniu nawiasów itp. Jest to zatem po strukturze leksykalnej i strukturze składniowej trzeci poziom abstrakcji, z jakiego chcemy teraz patrzeć na język programowania.

Do tej pory mówiliśmy, co się dzieje w pierwszych fazach kompilacji programu, do parsera włącznie. Parser generuje (*implicite* lub czasem *explicite*[1]) abstrakcyjne drzewo rozbioru programu. W kolejnych rozdziałach skryptu będzie nas interesować, co się dzieje dalej. Będziemy przy tym traktować program jako zbiór abstrakcyjnych drzew rozbioru i nie będziemy się zajmować jego składnią konkretną.

Składnia konkretna powinna być ściśle związana ze składnią abstrakcyjną języka. W szczególności powinien istnieć związek pomiędzy konkretnym drzewem wyprowadzenia programu z gramatyki i jego abstrakcyjnym drzewem rozbioru. Ułatwia to znacznie napisanie parsera.

Programy możemy przedstawiać formalnie jako termy stałe nad odpowiednio dobraną sygnaturą. Wówczas opis składni abstrakcyjnej sprowadzi się do podania pewnej sygnatury wielogatunkowej. Oczywiście *mówiąc* o termach (wyrażeniach) będziemy zapisywać je w postaci ciągu znaków, teraz jednak nasz zapis będzie należał do *metajęzyka*, tj. języka, *w którym mówimy* o termach, nie do języka termów. Będziemy przy tym używać pewnej notacji, np. infiksowej, lecz nie będzie ona przedmiotem naszego zainteresowania, tylko środkiem umożliwiającym mówienie o pewnych abstrakcyjnych obiektach.

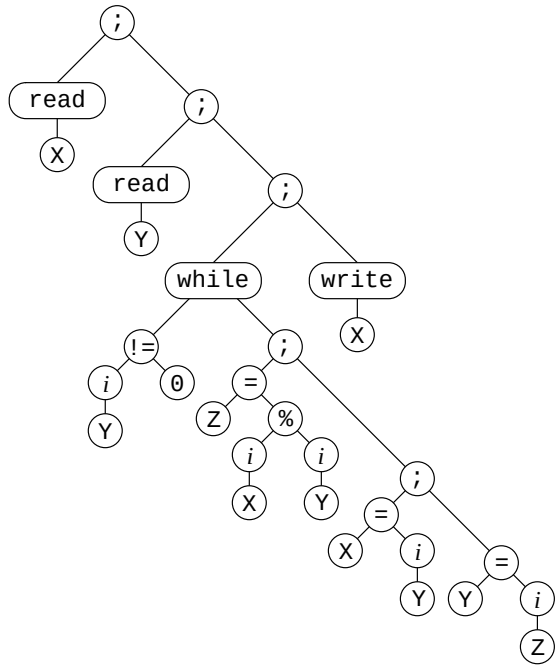
Do opisu języka D wykorzystamy zbiór czterech gatunków $\mathcal{S} = \{A, B, C, I\}$. Gatunek A będzie reprezentował wyrażenia arytmetyczne (*Arithmetic*), B — logiczne (*Boolean*), C — instrukcje (*Commands*), I zaś — identyfikatory (*Identifiers*). Sygnatura jest przedstawiona w tablicy 8.4 na poprzedniej stronie. Zauważmy, że zbiory $\Sigma_D^{A\to A}$ i $\Sigma_D^{A\times A\to A}$ nie są rozłączne, podobnie jak zbiory $\Sigma_D^{B\times C\to C}$ i $\Sigma_D^{B\times C\times C\to C}$. Ponieważ jednak symbol `-` występujący w dwóch pierwszych zbiorach ma w nich różną arność, podobnie jak symbol `if` występujący w dwóch ostatnich z wymienionych zbiorów, takie nadwerężenie notacji nie prowadzi do nieporozumień. Zbiory Σ_D^A i Σ_D^I są nieskończone. Wymieniliśmy tylko po kilka ich elementów. Zbiorów zmiennych nie będziemy na razie precyzować. Termy reprezentujące programy będziemy nazywać *abstrakcyjnymi drzewami rozbioru* tych programów. Ponieważ po lewej stronie symbolu przypisania może pojawić się tylko identyfikator, wyróżniliśmy identyfikatory jako osobny gatunek I. Z drugiej strony identyfikatory mogą pojawiać się po prawej stronie symbolu przypisania, tj. tam, gdzie pojawiają się termy gatunku A. W tym celu wprowadziliśmy specjalny symbol $i \in \Sigma_D^{I\to A}$, który niejako „przenosi" identyfikatory do gatunku A. Symbol i bywa nazywany *wyłuskaniem* (*dereferencją*). Mówimy o nim więcej w rozdziale 5. W niektórych językach wyłuskanie występuje również w składni konkretnej (np. w SML-u). W języku D, podobnie jak w C i Pascalu, wyłuskanie jest *niejawne* i sam identyfikator może pojawić się zarówno po lewej, jak i po prawej stronie symbolu przypisania.

Abstrakcyjne drzewo rozbioru programu z tablicy 3.10 w notacji prefiksowej jest przedstawione w tablicy 8.5 na następnej stronie. Wygodniej jednak potraktować symbole `;`, `=`, `!=` i `%` jako infiksowe (wymienione w kolejności priorytetów od najmniejszego do największego, przy czym `;` wiąże w prawo, `%` w lewo, zaś `=` i `!=` nie są łączne) i wówczas nasz term zapiszemy znacznie czytelniej:

[1] Czytelnik zechce zajrzeć do plików źródłowych kompilatora SML/NJ (również napisanego w SML/NJ). Parser SML/NJ *explicite* buduje strukturę danych będącą drzewem rozbioru programu.

```
;(read(X),
  ;(read(Y),
    ;(while(!=(i(Y), 0),
              ;(=(Z, %(i(X), i(Y))),
                ;(=(X, i(Y)),
                  =(Y, i(Z))))),
      write(X))))
```

Tablica 8.5. Abstrakcyjne drzewo rozbioru programu z tablicy 3.10 na stronie 90 w notacji prefiksowej



Rysunek 8.2. Abstrakcyjne drzewo rozbioru programu z tablicy 3.10 na stronie 90

stałe całkowite:	c	::=	`0` | `1` | `2` | ...
nazwy komórek pamięci:	$\mathcal{I}$	::=	`X` | `Y` | `Z` | ...
wyrażenia arytmetyczne:	$\mathcal{A}$	::=	c | $\mathcal{I}$ | $-\mathcal{A}$ | $\mathcal{A} \oplus \mathcal{A}$
wyrażenia logiczne:	$\mathcal{B}$	::=	`true` | `false` | $\mathcal{A} \oslash \mathcal{A}$ | `!` $\mathcal{B}$ | $\mathcal{B} \otimes \mathcal{B}$
operatory arytmetyczne:	$\oplus$	::=	`+` | `-` | `*` | `/` | `%`
operatory relacyjne:	$\oslash$	::=	`==` | `!=` | `<` | `>` | `<=` | `>=`
operatory logiczne:	$\otimes$	::=	`||` | `&&`
instrukcje:	$\mathcal{C}$	::=	`skip` | $\mathcal{I}$ `=` $\mathcal{A}$ | $\mathcal{C}$ `;` $\mathcal{C}$ | `if (`$\mathcal{B}$`)` $\mathcal{C}$ | `if (`$\mathcal{B}$`)` $\mathcal{C}$ `else` $\mathcal{C}$ | `while (`$\mathcal{B}$`)` $\mathcal{C}$

Tablica 8.6. Gramatyka abstrakcyjna języka D

`read(X) ; read(Y) ; while(`i`(Y) != 0, Z =` i`(X) % i(Y) ; X =` i`(Y) ; Y =` i`(Z)) ; write(X)`

W postaci graficznej ten sam term jest przedstawiony na rysunku 8.2 na poprzedniej stronie. W końcu możemy potraktować składnię konkretną języka jako jeden ze sposobów zapisu abstrakcyjnych drzew rozbioru programów.

Formalny opis języka za pomocą zbioru termów (drzew) ma tę wadę, że i tak musimy umawiać się co do sposobu zapisu (notacji) tych termów. Dlatego w praktyce składnię abstrakcyjną języka, podobnie jak składnię konkretną, opisujmy gramatyką bezkontekstową, zwaną *gramatyką abstrakcyjną*, często bardzo zwięzłą, ale niejednoznaczną. Zamiast jawnie podawać sygnaturę wielogatunkową opisujemy za pomocą gramatyki sposób zapisu abstrakcyjnych drzew rozbioru programu.

Abstrakcyjna gramatyka języka D jest przedstawiona w tablicy 8.6 i zawiera następujące kategorie syntaktyczne: stałe c, nazwy komórek pamięci $\mathcal{I}$, wyrażenia arytmetyczne $\mathcal{A}$, wyrażenia logiczne $\mathcal{B}$, operatory arytmetyczne $\oplus$, operatory relacyjne $\oslash$, operatory logiczne $\otimes$ i instrukcje $\mathcal{C}$. Mimo niejednoznaczności gramatyki zwykle z kontekstu jasno wynika sposób rozbioru wyrażeń. W szczególności jeśli zapiszemy wyrażenie $e_1 \oplus e_2$, to zakładamy, że operator $\oplus$ jest głównym operatorem w całym wyrażeniu (znajduje się w korzeniu jego abstrakcyjnego drzewa rozbioru). Zauważmy, że typom występującym w sygnaturze odpowiadają symbole nieterminalne (kategorie syntaktyczne) gramatyki.

8.7. Zadania

Zadanie 8.1. Rozważmy algebrę termów nad jednogatunkową sygnaturą $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, gdzie $\Sigma_0 = \{a, b, c\}$, $\Sigma_1 = \{f, g\}$, $\Sigma_2 = \{h, k\}$ i zbiorem zmiennych $\mathcal{X} = \{x, y, z, t, \ldots\}$. Znajdź najogólniejsze unifikatory dla następujących par termów:

1. $h(h(x, g(x)), g(x)) \stackrel{?}{=} h(h(y, z), g(y))$
2. $h(x, h(h(z, z), z)) \stackrel{?}{=} h(h(y, y), h(y, k(t, t)))$

3. $h(x, g(h(y, y))) \stackrel{?}{=} h(h(z, z), g(x))$
4. $k(h(a, x), k(x, y)) \stackrel{?}{=} k(h(y, a), z)$
5. $h(x, k(k(z, t), z)) \stackrel{?}{=} h(h(y, z), k(y, k(t, a)))$

Zadanie 8.2. Przez *rozmiar termu* będziemy rozumieć liczbę wystąpień symboli funkcyjnych i zmiennych w tym termie. *Rozmiarem równania* $t \stackrel{?}{=} s$ nazwiemy sumę rozmiarów termów t i s, zaś *rozmiarem podstawienia* $[x_i / t_i]_{i=1}^{n}$ nazwiemy sumę rozmiarów termów t_i. Pokaż, że istnieje równanie $t \stackrel{?}{=} s$ rozmiaru n, którego najogólniejszy unifikator ma rozmiar $2^{\Omega(n)}$.

Rozdział 9

Typy

Do uzupełnienia: Typy jako narzędzie kontroli poprawności programu. *Typefull programming* (Luca Cardelli). *Well-typed programs cannot go wrong.* Typowanie statyczne i dynamiczne.

9.1. Typy w SML-u

W SML-u kontrola typów przeprowadzana jest podczas kompilacji programu. System typów jest bardzo ścisły, nie pozwala na żadne niejawne konwersje itp. Z drugiej strony programista jest zwolniony z konieczności podawania typów zmiennych w programie — typy są odtwarzane przez komputer (choć programista może — np. dla czytelności — jawnie podać typ dowolnego wyrażenia).

9.1.1. Wyrażenia typowe

Wyrażenia typowe, to napisy złożone ze stałych typowych, np. `int`, `real`, `bool`, `char`, `string` itd., infiksowych operatorów: typu funkcyjnego `->` i krotki `*`, postfiksowych konstruktorów typowych, np. `list`, typów rekordowych `{}` oraz zmiennych typowych `'a`, `'b` itd. Przykładami wyrażeń typowych są:

```
int -> int * real -> int
{value : int, succ : int -> int}
(int * (int -> int)) list
'a list
'a * 'b list -> 'b
```

Formalnie składnia typów jest zadana w definicji języka SML [118] za pomocą reguł gramatycznych przedstawionych w tablicy 9.1 na następnej stronie, w której *tyvar* oznacza zmienne typowe (identyfikatory rozpoczynające się znakiem apostrofu „'"), *longtycon* to

ty	::=	*tyvar*	zmienna typowa
		{ [*tyrow*] }	typ rekordowy
		tyseq longtycon	konstruktor typu
		ty_1 * ... * ty_n	krotka, $n \geq 2$
		ty -> ty'	typ funkcyjny (R)
		(*ty*)	nawiasy do grupowania wyrażeń
tyrow	::=	*lab* : *ty* [, *tyrow*]	opis pól rekordu
tyseq	::=		puste
		ty	pojedynczy typ
		(ty_1, ... ty_n)	ciąg $n \geq 1$ typów

Tablica 9.1. Składnia wyrażeń typowych w języku Standard ML

identyfikator (być może prefiksowany nazwami modułów) oznaczający konstruktor typu (np. `list`) a *lab* to etykieta (identyfikator) pola rekordu. Operatory wymienione później w produkcji gramatyki wiążą słabiej. Napis (R) oznacza, że operator wiąże w prawo, a nawiasy [ ] oznaczają opcjonalny fragment produkcji. Poniżej omówimy dokładniej wszystkie wymienione rodzaje wyrażeń typowych.

W wielu różnych kontekstach aby wyrazić fakt, że x jest typu σ, będziemy pisać $x : \sigma$. Predefiniowane wartości proste mają jednoznacznie określone typy: liczby całkowite są typu `int`, np. `5 : int`, a liczby rzeczywiste — typu `real` (zatem, inaczej niż w Pascalu, `1` nie jest typu `real`), wartości boolowskie — typu `bool`, znaki — typu `char`, a łańcuchy — typu `string`.

Ponieważ funkcje są wartościami (mogą być elementami krotek, list, argumentami lub wynikami innych funkcji itp.) wprowadzono specjalną notację do opisu typów funkcji. Wyrażenie

$$\sigma \texttt{ -> } \tau$$

jest typem funkcji, której argument jest typu σ, a wynik — typu τ, np.

```
(fn x => x + 1) : int -> int
```

Infiksowy operator `->` wiąże w prawo, tj. σ `->` τ `->` ρ oznacza σ `->` (`(`τ `->` ρ`)`. Zachęcam do opuszczania zbędnych nawiasów. W wyrażeniu `(int -> bool) -> int` nawiasy są niezbędne (a wyrażenie to oznacza typ funkcji, która jako argument bierze funkcję przeprowadzającą liczby typu `int` w wartości typu `bool` i która zwraca wynik typu `int`). Po opuszczeniu nawiasów otrzymamy wyrażenie `int -> bool -> int` oznaczające typ funkcji, która bierze liczbę typu `int` i zwraca w wyniku funkcję, która przeprowadza wartości boolowskie w liczby całkowite.

Jeśli wyrażenie e, w którym występuje zmienna $x : \sigma$ ma typ τ, to `fn` x `=>` e ma typ σ `->` τ. Z drugiej strony, jeśli wyrażenie e_1, którego wartością jest funkcja, ma typ σ `->` τ, a wyrażenie e_2 jest typu σ, to typem wyniku aplikacji funkcji e_1 do argumentu e_2 jest τ,

tj. $(e_1 e_2) : \tau$. Dla przykładu:

```
Math.sin 1.0 : real
```

ponieważ `Math.sin : real -> real` oraz `1.0 : real`, zaś

```
(fn x => x + 1)(2+3) : int
```

gdyż `(fn x => x + 1) : int -> int` oraz `2+3 : int`.

Strukturalne typy danych, takie jak listy, rekordy i krotki, mogą mieć elementy dowolnych typów. Do oznaczenia krotki w wyrażeniach typowych służy gwiazdka `*`, która *nie jest łączna* i wiąże silniej od strzałki `->`. Typem krotki $(e_1, \ldots, e_n)$ jest σ_1 `*` $\ldots$ `*` σ_n, gdzie $e_i : \sigma_i$, np.:

```
  (1,true) : int * bool
((1,1),1) : (int * int) * int
  (1,1,1) : int * int * int
(1,(1,1)) : int * (int * int)
```

Ostatnie trzy przykłady wyjaśniają dlaczego `*` nie jest łączna.

Lista elementów typu σ ma typ σ `list`, np.:

```
                    [1] : int list
              [true,false] : bool list
["Cudak", "Oskar"] : string list
[Math.sin, Math.cos] : (real -> real) list
```

Postfiksowy, unarny operator `list` wiąże silniej od operatorów `->` i `*`, dlatego w drugim przykładzie nawiasy są niezbędne. Zauważmy, że identyfikator `list` nie jest nazwą typu — jest unarnym operatorem, za pomocą którego można tworzyć nazwy typów, takie jak `int list`, `(int -> int) list` itp.

Typem rekordu $\{l_1 = e_1, \ldots, l_n = e_n\}$ jest $\{l_1 : \sigma_1, \ldots l_n : \sigma_n\}$, gdzie $e_i : \sigma_i$, np.:

```
          {re=1.5, im=3.5} : {re:real, im:real}
{dzien=22, miesiac=1, rok=1969} : {dzien:int, miesiac:int, rok:int}
      {marka="fiat", cena=2300} : {marka:string, cena:int}
```

9.1.2. Nadawanie nazw typom danych

W SML-u dowolnej wartości można nadać nazwę („drugie imię") za pomocą deklaracji `val`. Podobnie można nazywać typy. Służy do tego tzw. *niegeneratywna deklaracja typu* postaci

$$\texttt{type}\ \vec{\alpha}\ T = \sigma$$

w której $\vec{\alpha}$ jest opcjonalnym ciągiem parami różnych zmiennych typowych (o zmiennych typowych mówimy w podrozdziale 9.3) oddzielonych przecinkami i ujętych w nawiasy (jeśli jest to pojedyncza zmienna, nawiasy można opuścić), T jest identyfikatorem, a σ — wyrażeniem typowym. Wszystkie zmienne typowe występujące w wyrażeniu σ muszą wystąpić w ciągu $\vec{\alpha}$. Dla przykładu

```
type weight = int
type coord  = int * int
type translation = coord -> coord
type 'a epi = 'a -> 'a
type ('a,'b) map = 'a -> 'b
type ('a,'b) prod = {x:'a, y:'b}
```

Wprowadzonych w ten sposób nazw typów możemy używać w programach we wszelkich kontekstach, gdzie może pojawić się wyrażenie typowe. Dzięki temu programy stają się bardziej czytelne. Jeżeli na przykład punkt na płaszczyźnie reprezentujemy w postaci pary liczb całkowitych i chcemy zaprogramować jednokładność o skali `2.0` i środku w punkcie `(0.0,0.0)`, możemy napisać:

```
- type point = real * real
= fun zoom ((x,y) : point) : point = (2.0 * x, 2.0 * y);
type point = real * real
val zoom = fn : point -> point
```

W odpowiedzi system potwierdza, że typ `point` znaczy dla niego to samo, co `real*real` i, skoro jawnie tego zażądaliśmy, używa nazwy `point` wypisując typ funkcji `zoom`. Dzięki temu jej typ niesie ze sobą dużo więcej informacji na temat jej zamierzonego działania niż anonimowe wyrażenie `real * real -> real * real`.

Deklaracja `type` *nie definiuje* nowego typu, wprowadza jedynie nową nazwę dla typu już istniejącego. Zgodnie z zasadą przezroczystości każde wystąpienie nowej nazwy typu można zastąpić wyrażeniem z nim związanym (i odwrotnie). System używa nazwy typu tylko tam, gdzie tego jawnie zażądamy:

```
- val p1 = (1.0,1.0);
val p1 = (1.0,1.0) : real * real
- val p2 : point = (1.0,1.0);
val p2 = (1.0,1.0) : point
- zoom p1;
val it = (2.0,2.0) : real * real
- zoom p2;
val it = (2.0,2.0) : real * real
```

9.2. Rekonstrukcja typów

W starszych językach programowania (np. w Pascalu) typy muszą występować *explicite* w programie. Każda zmienna, wartość, funkcja itp. musi mieć jawnie podany typ. Zadaniem kompilatora takiego języka jest jedynie sprawdzenie, czy typy podane przez programistę są poprawne. Jest to tzw. *kontrola typów* (ang. *type checking*). Wiele nowoczesnych języków zwalnia programistę z konieczności podawania typów. Wówczas kompilator musi je odtworzyć analizując program, tj. dokonać *rekonstrukcji typów* (ang. *type reconstruction*).

Nieobecność typów w tekście programu nie oznacza, że kompilator nie wykrywa błędów typowych. Typy te występują w programie *implicite*. Rozważmy na przykład program w SML-u:

```
fun f x = x + 1
```

Możemy przeprowadzić następujące rozumowanie: ponieważ w treści funkcji do zmiennej `x` dodaje się stałą `1`, to `x` musi być typu `int`, podobnie jak całe wyrażenie `x + 1`. Zatem `f : int -> int`. Typ funkcji `f` można więc odtworzyć, mimo iż nie jest on podany *explicite* w programie.

Ponieważ niemal każda wartość w programie ma jednoznacznie określony typ (wyjątkiem jest kilka stałych i operatorów przeciążonych), to kompilator stosując opisane w podrozdziale 9.1.1 reguły przypisywania typów wyrażeniom potrafi sam odtworzyć ich typ i programista jawnie nie musi go podawać.

W SML-u po dowolnym wyrażeniu w programie i po dowolnym wzorcu można napisać tzw. *zawężenie typu* (ang. *type constraint*), tj. dwukropek `:` i wyrażenie typowe, np.:

```
(1)  fun f (x:int) = x+1
(2)  fun f x :int = floor x
(3)  fun f x = (x:int) + 1
(4)  fun f x = x+1 :int
```

Kompilator sprawdzi, czy wyznaczony przezeń typ zgadza się z podanym zawężeniem. W deklaracji (2) zawężenie typu „`:int`” dotyczy wyniku funkcji, nie jej argumentu `x`. W SML-u możemy więc typów nie podawać wcale:

```
fun f x = x + 1
```

lub też np. pisać tak, jak w Pascalu:

```
fun f (x : int) : int = x + 1
```

W obu przypadkach kompilator sam wyznaczy typy występujących w programie obiektów. W drugim przypadku sprawdzi dodatkowo, czy podane przez programistę zawężenia zgadzają się z typami zrekonstruowanymi. Taki mechanizm rekonstrukcji typów, zwany *częściową rekonstrukcją typów* (ang. *partial type reconstruction*) występuje w wielu nowoczesnych językach programowania (Haskell, Concurrent Clean itp). Zwalnia on programistę z konieczności żmudnego wypisywania typów w programie, przez co program staje się krótszy i czytelniejszy, choć pozwala na ich podanie tam, gdzie programista uzna to za właściwe (w celu np. lepszego udokumentowania programu lub wymuszenia typu innego niż wyznaczony przez kompilator, o czym jest mowa w następnym podrozdziale). Używany powszechnie algorytm rekonstrukcji typów, tzw. *algorytm W* został wynaleziony przez Robina Milnera [38, 115] (por. podrozdział 9.4).

9.3. Polimorfizm

Do uzupełnienia: Treść tego podrozdziału jest jedynie naszkicowana i wymaga rozszerzenia o dalsze przykłady i komentarze.

9.3.1. Polimorfizm parametryczny

Rozważmy funkcję identyczności w SML-u:

```
fun Id x = x
```

Jaki typ powinien wyznaczyć kompilator dla funkcji `Id`? Może ona mieć typ `int -> int`, `bool -> bool` i ogólniej σ `->` σ, dla *dowolnego* typu σ, bowiem dla wartości *każdego* typu umiemy wygenerować kod wynikowy funkcji, która pobiera argument tego typu i zwraca go w wyniku. Organizując odpowiednio reprezentację danych w pamięci maszyny możemy wygenerować *jeden wspólny* kod wynikowy dla funkcji `Id`, dobry dla argumentów każdego typu. Zamiast więc definiować szereg funkcji

```
fun intId (x : int) : int = x
fun boolId (x : bool) : bool = x
fun intListId (x : int list) : int list = x
       ⋮
```

chcielibyśmy zdefiniować tylko jedną funkcję `Id` i umieć opisać jej typ. Każdy typ postaci σ `->` σ, gdzie σ jest dowolnym typem jest dobrym typem dla `Id` i odwrotnie, każdy adekwatny typ dla funkcji `Id` musi być tej postaci. Wygodnie jest zatem rozszerzyć notację dla wyrażeń typowych tak, by móc napisać, że typem funkcji `Id` jest α `->` α, gdzie α nie jest nazwą konkretnego typu, tylko *zmienną* przebiegającą wszystkie możliwe typy. Dlatego chciałoby się napisać `Id :` α `->` α. W tym celu wprowadzono do systemu typów SML-a *zmienne typowe* (tj. zmienne przebiegające typy), które mogą występować w wyrażeniach typowych. Ponieważ w kodzie ASCII nie ma greckich liter, przyjęto oznaczać je używając małych liter z początku alfabetu, poprzedzonych apostrofem: `'a`, `'b`, `'c` itd. Identyfikator `'a` często czyta się „alfa”, a `'b` — „beta”. Na papierze korzysta się z obu notacji (α zamiast `'a`), a czasem też pisze matematyczną strzałkę ($\rightarrow$) zamiast `->`. Wracając do naszego przykładu, kompilator odpowiada:

```
- fun id x = x;
val id = fn : 'a -> 'a
```

wskazując, że funkcja `Id` bierze argument dowolnego typu `'a` i zwraca wartość tego samego typu. Funkcję `Id` możemy teraz zaaplikować zarówno do wartości typu `int` jak i `bool`:

```
- id 1;
val it = 1 : int
- id true;
val it = true : bool
```

a nawet do samej siebie:

```
- id id;
val it = fn : 'a -> 'a
```

W ostatnim przypadku, skoro `Id` jest typu `'a -> 'a`, to w szczególności jest też typu `('a -> 'a) -> ('a -> 'a)`, bo skoro jest funkcją, która bierze argument dowolnego typu i zwraca wartość tego samego typu, to w szczególności jest funkcją która bierze funkcję która bierze argument dowolnego typu i zwraca wartość tego samego typu jako argument i zwraca wynik tego samego typu. `Id` może być zatem zaaplikowana do funkcji która bierze argument dowolnego typu i zwraca wartość tego samego typu, tj. do samej siebie. W wyniku dostajemy funkcję która bierze argument dowolnego typu i zwraca wartość tego samego typu, tj. funkcję typu `'a -> 'a`.

Polimorfizm parametryczny pojawia się wówczas, gdy funkcja nie wnika w strukturę swojego argumentu. Jego typ, podobnie jak i wartość, są więc nieistotne i mogą być dowolne. Dlatego, w odróżnieniu od przeciążania (zob. podrozdział 9.3.2), w którym treści funkcji dla różnych typów nie muszą być ze sobą związane, w przypadku polimorfizmu parametrycznego kod funkcji jest jeden, przetwarza jedynie „niedospecyfikowane" wartości.

Typ zawierający zmienne nazywamy *polimorficznym*, typ nie zawierający zmiennych — *monomorficznym*. Wartości polimorficzne (tj. będące polimorficznego typu) pojawiają się w programowaniu funkcjonalnym na każdym kroku. M. in. dzięki nim programy w SML-u są znacznie krótsze od ich pascalowych czy C-owych odpowiedników: ten sam fragment kodu może być wykorzystywany w różnych kontekstach i do różnych celów. Np. funkcja sortująca listy może mieć typ:

```
sort : ('a * 'a -> bool) -> 'a list -> 'a list
```

i brać jako parametr binarną relację porównującą elementy typu `'a` i zwracać w wyniku konkretną funkcję sortującą listy elementów typu `'a`. Dzięki temu raz napisany program może być używany do sortowania list dowolnego typu, a jednocześnie zachowana jest pełna kontrola typów (por. rozwiązania tego problemu w bibliotekach Pascala lub C, które polegają zwykle na wyłączeniu kontroli typów i oszukaniu kompilatora).

Idea polimorfizmu parametrycznego świetnie współgra z mechanizmem rekonstrukcji typów. Jednak w języku typowanym *explicite* polimorfizm parametryczny również jest możliwy. Przykładem są *szablony* (ang. *templates*) w języku C++. Można w nim bowiem napisać deklarację postaci

```
template <class T>
   T Id (T x) {
      return x;
   }
```

która definiuje szablon, opisujący klasę funkcji

```
int Id (int x) { return x; }
float Id (float x) { return x; }
char* Id (char* x) { return x; }
   ⋮
```

O tym, która z tych funkcji zostanie użyta, decyduje kontekst, tj. typ argumentu faktycznego funkcji `Id`. Jednak, w odróżnieniu od SML-a, dla każdego typu jest generowany *osobny* kod

wynikowy. Szablony oszczędzają więc miejsce w kodzie źródłowym, jednak nie w pliku wynikowym. Szablonu nie można też skompilować i udostępniać w bibliotece. Można jedynie udostępniać jego kod źródłowy kompilowany za każdym razem, gdy z niego korzystamy. Szablony w C++ są więc w zasadzie namiastką polimorfizmu parametrycznego.

Polimorfizm parametryczny a skutki uboczne. Polimorfizm parametryczny jest dosyć trudno pogodzić w jednym języku z występowaniem skutków ubocznych, czyli z programowaniem imperatywnym.

Przykład 9.1. Rozważmy następujące deklaracje w SML-u:

```
val x = ref []
fun f y = y + hd(!x)
val _ = x := [true]
val _ = f 1
```

Skoro `[]` : α `list`, to `x` : (α `list`) `ref` (tj. `x` jest wskaźnikiem do listy elementów dowolnego typu). W takim razie `f : int` $\rightarrow$ `int`. Skoro `x` : (α `list`) `ref`, to w szczególności można mu przypisać listę `[true]`. Ale wówczas podczas poprawnego typowo wywołania `f 1` dojdzie do próby obliczenia wyrażenia `1 + true`, które jest pozbawione sensu.

W każdym języku, który posiada polimorficzne konstrukcje imperatywne na system typów narzuca się pewne dodatkowe ograniczenia. Rozwiązań jest wiele a ich opis wykracza poza ramy bieżącego wykładu. Powiemy jedynie, że w obecnej definicji języka Standard ML przyjęto tzw. *regułę ograniczenia polimorfizmu do wartości* (ang. *value restriction*), która powoduje, że typy niektórych wyrażeń zamiast spodziewanych zmiennych typowych zawierają tzw. *atrapy typów* (ang. *dummy types*). Na przykład w kontekście deklaracji:

```
- fun f x y = x;
val f = fn : 'a -> 'b -> 'a
```

dla funkcji g zadanej deklaracją

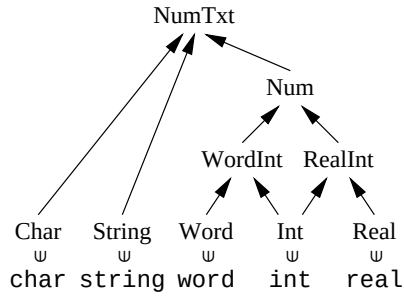
```
val g = f 5
```

wyprowadzilibyśmy typ `'a -> int`. Tymczasem system odpowiada:

```
- val g = f 5;
Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val g = fn : ?.X1 -> int
```

Atrapa typu `?.X1` jest stałą typową nazywającą nie istniejący typ. Funkcji g nie można zaaplikować do żadnego argumentu. Rozwiązaniem tego problemu jest zdefiniowanie funkcji g w następujący sposób:

```
- fun g x = f 5 x;
val g = fn : 'a -> int
```



Rysunek 9.1. Klasy typów w standard ML-u

Reguła ograniczenia polimorfizmu do wartości powoduje, że program z przykładu 9.1 jest odrzucany przez kompilator:

```
- val x = ref [];
Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val x = ref [] : ?.X1 list ref
- fun f y = y + hd(!x);
Error: overloaded variable not defined at type
  symbol: +
  type: ?.X1
```

9.3.2. Przeciążanie

Do uzupełnienia: Polimorfizm *ad hoc*. Operatory przeciążone w Adzie i C++. Przeciążanie a rekonstrukcja typów. Klasy typów w Haskellu i Concurrent Cleanie.

Identyfikatory przeciążone w Standard ML-u. W Standard ML-u predefiniowano 12 operatorów przeciążonych, których lista znajduje się w tablicy 9.2. Są to jedyne operatory przeciążone, programista nie może przeciążać żadnych identyfikatorów. Wykorzystanie operatora przeciążonego do nazwania nowej wartości powoduje, że przestaje on być przeciążony. Standard języka przewiduje 9 klas typów, przedstawionych na rysunku 9.1. Każda z pięciu klas bazowych Char, String, Word, Int i Real powinna zawierać co najmniej jeden typ (opowiednio `char`, `string`, `word`, `int` i `real`), jednak klasy te w konkretnej implementacji mogą też zawierać inne typy (np. `word32`, `word64` opisujące liczby całkowite bez znaku pojedynczej i podwójnej długości itp).

Prefiksowe:

`abs` : RealInt → RealInt
`~` : RealInt → RealInt

Infiksowe, priorytetu 7, łączące w lewo:

`div` : WordInt → WordInt
`mod` : WordInt → WordInt
`*` : Num → Num
`/` : Real → Real

Infiksowe, priorytetu 6, łączące w lewo:

`+` : Num → Num
`-` : Num → Num

Infiksowe, priorytetu 4, łączące w lewo:

`<` : NumTxt → NumTxt
`>` : NumTxt → NumTxt
`<=` : NumTxt → NumTxt
`>=` : NumTxt → NumTxt

Tablica 9.2. Identyfikatory przeciążone w Standard ML-u

9.3.3. Typy równościowe w SML-u

Niektóre identyfikatory w SML-u nazywają jedną konkretną wartość ustalonego typu (tj. są monomorficzne), inne (przeciążone) nazywają dwie lub trzy wartości różnych typów, inne w końcu (polimorficzne) nazywają całe kolekcje wartości różnych typów. Operatory relacyjne równości = i różności <> mają specjalny status: nie mogą być polimorficzne, ponieważ nie dla wszystkich typów daje się określić relację równości: np. nie dla funkcji (problem sprawdzenia, czy dwa algorytmy obliczają wartości tej samej funkcji jest nierozstrzygalny). Nadto, w niezgodzie z ideą polimorfizmu parametrycznego, dla każdego typu kod funkcji = musi być inny. Z drugiej strony równość określona jest dla tak wielu typów, że byłoby niewygodnie przeciążać identyfikatory <> i =. Dlatego wprowadzono w SML-u specjalne zmienne typowe, tzw. *zmienne równościowe*, które przebiegają jedynie *typy równościowe*, tj. typy dla których określono relację równości. Oznacza się je podwajając apostrof na początku zmiennej: `''a`, `''b` itd., a na papierze pisze się czasem $\alpha^{=}$, $\beta^{=}$ itd. Mamy więc:

```
=  : ''a * ''a -> bool
<> : ''a * ''a -> bool
```

Typy równościowe to wszystkie typy (również listy, krotki, rekordy itp.) które nie zawierają funkcji. Kompilator odrzuci porównanie dwóch wartości jako niepoprawne, jeśli nie mają one typów równościowych:

```
- fun f x = [x] = nil;
```

```
val f = fn : ''a -> bool
- f (fn x => x);
Error: operator and operand don't agree [equality type required]
  operator domain: ''Z
  operand:         'Y -> 'Y
  in expression:
    f (fn x => x)
```

Trwa ciągły spór zwolenników i przeciwników uczynienia typu `real` typem równościowym. Był on typem równościowym w standardzie SML'90 i w kompilatorze SML/NJ 0.93. Z wielu (również filozoficznych) powodów porównywanie wartości typu `real` zostało zabronione w standardzie SML'97 i w nowszych wersjach kompilatora SML/NJ. Ta decyzja jest zgodna z normą IEEE dotyczącą maszynowej reprezentacji liczb zmiennoprzecinkowych. Wystarczy zadać kompilatorowi SML/NJ 0.93 proste pytanie:

```
- 1.2 - 1.0 = 0.2;
val it = false : bool
```

by przekonać się dlaczego. Błędy zaokrągleń powodują, że odpowiedzi maszyny są bezsensowne. Natomiast w bezbłędnej arytmetyce rzeczywistej porównywanie jest nierozstrzygalne. Z tych samych powodów typ `real` nie posiada konstruktorów, nie można zatem badać jego wartości we wzorcach. Zamiast warunku `x=0.0` należy raczej używać warunku `abs(x)<epsilon`, gdzie `epsilon` jest dostatecznie małą liczbą dodatnią (np. `1e 10`). Jeżeli nawet uzasadnimy teoretycznie, że w pewnym procesie iteracyjnym po wykonaniu skończonej liczby kroków zmienna `x` przyjmie wartość 0, to wykonywanie iteracji tak długo, aż będzie spełniony warunek `x=0.0` może się ze względu na błędy zaokrągleń nigdy nie zakończyć. Lepiej wówczas testować warunek `abs(x)<epsilon` dla stosownie dobranej dodatniej wartości `epsilon`, tak, by wartość bezwzględna każdej niezerowej wartości parametru `x` była od niej większa. Operatory porównania są zdefiniowane dla liczb rzeczywistych, choć niestety ich użycie może prowadzić do takich samych nonsensów:

```
- 0.2 <= 1.2 - 1.0;
val it = false : bool
- 0.2 > 1.2 - 1.0;
val it = true : bool
```

Do uzupełnienia:

9.3.4. Dziedziczenie

9.4. Algorytm rekonstrukcji typów w Standard ML-u

Reguły wyprowadzania typów dla programów w Standard ML-u zostały nieformalnie opisane w podrozdziale 9.1. Obecnie opiszemy ideę algorytmu *W* Milnera rekonstrukcji typów, będącego częścią składową każdego kompilatora języka Standard ML. Algorytm ten

$$
\begin{array}{rcl}
\mathcal{I} & ::= & x \mid y \mid z \mid \ldots \\
\mathcal{N} & ::= & 0 \mid 1 \mid 2 \mid \ldots \\
\mathcal{E} & ::= & \mathcal{N} \mid \texttt{true} \mid \texttt{false} \mid \mathcal{I} \mid \mathcal{E} + \mathcal{E} \mid (\mathcal{E}, \mathcal{E}) \mid \texttt{fn}\ \mathcal{I}\ \texttt{=>}\ \mathcal{E} \mid \mathcal{E}\ \mathcal{E} \mid \\
 & & \texttt{let val}\ \mathcal{I} = \mathcal{E}\ \texttt{in}\ \mathcal{E}\ \texttt{end} \mid \texttt{let val rec}\ \mathcal{I} = \mathcal{E}\ \texttt{in}\ \mathcal{E}\ \texttt{end} \\
\mathcal{V} & ::= & \alpha \mid \beta \mid \gamma \mid \ldots \\
\mathcal{T} & ::= & \mathcal{V} \mid \texttt{int} \mid \texttt{bool} \mid \mathcal{T} * \mathcal{T} \mid \mathcal{T} \to \mathcal{T}
\end{array}
$$

Tablica 9.3. Składnia abstrakcyjna języka Mini-ML

zbudujemy jedynie dla niewielkiego fragmentu SML-a, którego składnia abstrakcyjna jest przedstawiona w tablicy 9.3 i który nazwaliśmy Mini-ML-em. W ten sposób wydzieliliśmy istotną część języka ze zbioru dziesiątków konstrukcji składniowych, których opis jedynie zaciemniałby istotę algorytmu.

Programy w Mini-ML-u są wyrażeniami należącymi do kategorii składniowej $\mathcal{E}$. Wyrażenie może być zmienną (elementem kategorii składniowej $\mathcal{I}$), stałą całkowitoliczbową bez znaku (elementem kategorii $\mathcal{N}$), stałą logiczną `true` lub `false`, parą (e_1, e_2) wyrażeń e_1 i e_2, abstrakcją funkcyjną, aplikacją funkcji do argumentu, wyrażeniem `let` zawierającym pojedynczą definicję nierekurencyjną lub wyrażeniem `let` zawierającym pojedynczą definicję rekurencyjną. Wyrażenie typowe $\mathcal{T}$ (nie występujące w programach, ale wypisywane przez kompilator) może być zmienną typową (elementem kategorii $\mathcal{V}$), stałą typową `int` lub `bool`, lub typem funkcyjnym $\sigma \to \tau$, gdzie σ jest typem argumentu, zaś τ typem wyniku funkcji.

Reguły wyprowadzania typów dla programu, krócej *reguły typowania* jest wygodnie opisać w postaci formalnego systemu wnioskowania, tzw. *systemu typów*. Najpierw opiszemy najprostszy system typów, zwany *monomorficznym*. Następnie zbadamy jego wady i zdefiniujemy dla Mini-ML-a inny system, tzw. *system typów z polimorfizmem parametrycznym*. Będzie on odpowiadał systemowi typów występującemu w „prawdziwym" języku SML. Na koniec rozważymy jedno z jego rozszerzeń, tzw. system z *rekursją polimorficzną*.

9.4.1. Monomorficzny system typów

Na oznaczenie faktu, że dla wyrażenia e wyprowadziliśmy typ τ piszemy formułę postaci $\vdash e : \tau$ zwaną *tezą typową* lub *sądem typowym* (ang. *type judgement* lub *typing*). Mówimy wówczas, że wyrażenie *e posiada typ* τ.

Opiszemy teraz zbiór aksjomatów i reguł wnioskowania, za pomocą których będziemy udowadniać sądy typowe. Jest bardzo wygodnie, jeśli system typów jest *sterowany składnią* (ang. *syntax directed*), tj. taki, w którym reguły wyprowadzania typów odpowiadają regułom budowania wyrażeń w gramatyce abstrakcyjnej języka. Każdej produkcji gramatyki abstrakcyjnej jest przyporządkowana dokładnie jedna reguła typowania. Reguły te są postaci: „jeżeli wyrażenie e składa się z podwyrażeń $e_1, \ldots, e_n$ i ustaliliśmy, że typem wyrażenia e_i jest τ_i,

to typem wyrażenia e jest τ, zbudowany z typów τ_i w odpowiedni sposób". Dla przykładu jeżeli ustaliliśmy, że wyrażenie e_1 ma typ $\sigma \to \tau$, tj. $\vdash e_1 : \sigma \to \tau$, zaś wyrażenie e_2 jest typu σ, tj. $\vdash e_2 : \sigma$, to wynik aplikacji funkcji e_1 do argumentu e_2 jest typu τ, tj. $\vdash e_1 e_2 : \tau$. Dla aplikacji funkcji do argumentu powinniśmy mieć zatem w systemie typów regułę (za chwilę zobaczymy, że w istocie jest ona nieco ogólniejsza):

$$\frac{\vdash e_1 : \sigma \to \tau \quad \vdash e_2 : \sigma}{\vdash e_1 e_2 : \tau}$$

Ponieważ zmienne w programie mogą być dowolnego typu, pojawia się problem, jakiej postaci powinien być aksjomat dla zmiennych. Moglibyśmy spróbować napisać schemat aksjomatu $\vdash x : \sigma$ dla dowolnego typu σ, jednak jeżeli zmienna występuje w wyrażeniu kilkakrotnie, to wszystkie jej wystąpienia powinny mieć ten sam typ. W systemie typów musi zatem istnieć pewien mechanizm zapamiętywania, jakiego typu są zmienne występujące w typowanym wyrażeniu. Do tego celu służy tzw. *kontekst*, tj. skończony zbiór par postaci $\Gamma = \{x_i : \sigma_i\}$, gdzie x_i są zmiennymi, zaś σ_i typami i w którym wszystkie zmienne x_i są parami różne. Napis $\Gamma, x : \sigma$ oznacza kontekst $\Gamma \cup \{x : \sigma\}$. Nawiasy $\{\}$ czesto pomijamy, zapisując kontekst jako ciąg oddzielonych przecinkami założeń typowych. Tezy systemu typów są postaci $\Gamma \vdash e : \sigma$, co znaczy „jeżeli zmienne mają takie typy, jak podano w kontekście Γ, to wyrażenie e (zawierające wolne wystąpienia tych zmiennych) ma typ σ". Zamiast $\emptyset \vdash e : \sigma$ pisze się $\vdash e : \sigma$. Jak zaraz zobaczymy, tezy takiej postaci są wyprowadzalne w systemie jedynie dla wyrażeń e nie zawierających zmiennych wolnych. Aksjomat dla zmiennych ma postać

$$\Gamma, x : \sigma \vdash x : \sigma$$

co znaczy „jeśli przyjęliśmy (założyliśmy), że typem zmiennej x jest σ, to typem zmiennej x jest σ". Aksjomat ten pozwala skorzystać (być może wielokrotnie) z założenia na temat typu zmiennej x. Ponieważ w kontekście nie może istnieć więcej niż jedno założenie o typie tej samej zmiennej, kontekst przechowuje jednoznaczną informację o typie każdej zmiennej wolnej występującej w typowanym wyrażeniu. Reguła typowania aplikacji ma w naszym systemie typów postać

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau}$$

Zauważmy, że w obu przesłankach oraz w konkluzji reguły występuje *ten sam* kontekst. Dzięki temu każde wystąpienie tej samej zmiennej będzie mieć przypisany ten sam typ, np.:

$$\frac{\dfrac{\overline{x : \texttt{int},\ f : \texttt{int} \to (\texttt{int} \to \texttt{bool}) \vdash f : \texttt{int} \to (\texttt{int} \to \texttt{bool})} \quad \overline{x : \texttt{int}, \ldots \vdash x : \texttt{int}}}{x : \texttt{int},\ f : \texttt{int} \to (\texttt{int} \to \texttt{bool}) \vdash f\ x : \texttt{int} \to \texttt{bool}}}{x : \texttt{int},\ f : \texttt{int} \to (\texttt{int} \to \texttt{bool}) \vdash (f\ x)\ x : \texttt{bool}}$$

gdzie, z braku miejsca, nie wypisaliśmy całego kontekstu w jednym z aksjomatów.

Ponieważ typ stałych logicznych i liczbowych jest ustalony i nie zależy od kontekstu, system typów zawiera aksjomaty $\Gamma \vdash \texttt{true} : \texttt{bool}$, $\Gamma \vdash \texttt{false} : \texttt{bool}$, $\Gamma \vdash 0 : \texttt{int}$, $\Gamma \vdash 1 : \texttt{int}$, itd. Dla przykładu, aksjomat $\Gamma \vdash 2 : \texttt{int}$ mówi, że „niezależnie od tego,

$$\frac{}{\Gamma \vdash c : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e : \sigma \to \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma\ \texttt{*}\ \tau} \qquad \frac{\Gamma x : \sigma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val rec}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

Tablica 9.4. Monomorficzny system typów dla Mini-ML-a

jakie założenia poczyniliśmy o typach zmiennych występujących w programie, stała 2 ma typ `int`".

Pełen zestaw aksjomatów i reguł wnioskowania monomorficznego systemu typów dla Mini-ML-a jest przedstawiony w tablicy 9.4. Reguły typowania sumy wyrażeń i pary nie wymagają obszernego komentarza. Pierwsza z nich mówi, że jeśli wyrażenia e_1 i e_2 mają typ `int`, wówczas wyrażenie $e_1 + e_2$ jest poprawne (w sensie kontroli typów) i również ma typ `int`. Ponieważ jest to jedyna reguła, za pomocą której można otypować sumę wyrażeń, jeśli dla któregoś z wyrażeń nie da się wyprowadzić typu `int`, to *nie istnieje* żaden dowód, który w konkluzji miałby tezę postaci $\Gamma \vdash e : \sigma$ dla *żadnego* kontekstu Γ i *żadnego* typu σ. Dla przykładu nie istnieje żadna taka teza dla wyrażenia $e = 1 + \texttt{true}$. Na tym polega kontrola poprawności programu: kompilator odrzuca wyrażenie $1 + \texttt{true}$ jako niepoprawne, ponieważ nie potrafi wyprowadzić dla niego żadnego typu:

```
- 1+true;
Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * bool
  in expression:
    1 + true
```

Podobnie dla wyrażenia (e_1, e_2) można wyprowadzić typ σ `*` τ wtedy i tylko wtedy, gdy dla wyrażenia e_1 można wyprowadzić typ σ, zaś dla wyrażenia e_2 typ τ. Reguła

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e : \sigma \to \tau}$$

typowania abstrakcji funkcyjnej mówi, że jeżeli wyrażenie e ma typ τ przy założeniu, że występująca w nim zmienna x ma typ σ, to funkcja `fn` x `=>` e przyporządkowująca argumentowi x wynik e ma typ $\sigma \to \tau$. Zmienna x nie ma wolnych wystąpień w wyrażeniu `fn` x `=>` e, opisując jego typ nie musimy zatem zakładać niczego o jej typie. Dlatego kontekst konkluzji reguły typowania aplikacji różni się od kontekstu jej przesłanki tym, że nie

zawiera założenia o typie zmiennej x. W „oszczędnym” dowodzie kontekst zawiera wyłącznie założenia o typach zmiennych, które mają wolne wystąpienia w typowanym wyrażeniu. Kompletny poprawny program nie zawiera zmiennych wolnych (w przeciwnym razie pojawia się błąd kompilacji „niezadeklarowany identyfikator”), jeśli jest więc poprawny pod względem typów, to można dla niego wyprowadzić tezę typową z pustym kontekstem.

Ponieważ założyliśmy, że kontekst nie może zawierać dwóch założeń o tej samej zmiennej, żadne wiążące wystąpienie zmiennej nie może występować w zasięgu innego wiążącego wystąpienia tej samej zmiennej, np. wyrażenie `fn` x `=>` `fn` x `=>` x, choć poprawne, „nie pasuje” do naszego systemu. Zakładamy, że w takich sytuacjach będziemy dokonywać przemianowania zmiennych i wyprowadzać typ dla wyrażenia, w którym wszystkie zmienne są różne, np. dla `fn` x `=>` `fn` y `=>` y. W kompilatorach kontekst zwykle reprezentuje się nie jako zbiór, tylko stos, na który odkładane są założenia typowe w takiej samej kolejności, w jakiej miejsca wiązania zmiennych są zagnieżdżone w drzewie rozbioru programu. Wówczas można dopuścić *przesłanianie zmiennych* w programie, ponieważ założenie typowe o danej zmiennej wyszukuje się począwszy od wierzchołka stosu i kolejne założenia typowe o tej samej zmiennej przesłaniają poprzednie w takiej samej kolejności, w jakiej miejsca wiązania zmiennej występują w programie.

Reguły typowania obu wersji wyrażenia `let` również są dosyć naturalne. W regule

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end} : \tau}$$

wyrażenie e_2 może zawierać wolne wystąpienia zmiennej x. Znaczenie całego wyrażenia `let` jest takie, jak gdybyśmy wstawili w wyrażeniu e_2 w miejsce każdego wystąpienia zmiennej x wyrażenie e_1:

$$(\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) \quad = \quad e_2[x/e_1]$$

Zatem jeżeli dla wyrażenia e_1 wyprowadziliśmy typ σ i przy założeniu, że zmienna x ma właśnie ten typ σ wyprowadzimy dla wyrażenia e_2 typ τ, to całe wyrażenie `let` również ma typ τ. Dla przykładu

$$\dfrac{\dfrac{\dfrac{}{y : \texttt{int} \vdash y : \texttt{int}} \quad \dfrac{}{y : \texttt{int} \vdash y : \texttt{int}}}{y : \texttt{int} \vdash y + y : \texttt{int}} \quad \dfrac{\dfrac{}{y : \texttt{int}, x : \texttt{int} \vdash x : \texttt{int}} \quad \dfrac{}{\ldots x : \texttt{int} \vdash x : \texttt{int}}}{y : \texttt{int}, x : \texttt{int} \vdash (x, x) : \texttt{int * int}}}{y : \texttt{int} \vdash \texttt{let val } x = y + y \texttt{ in } (x, x) \texttt{ end} : \texttt{int * int}}$$

Wyrażenie

$$\texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}$$

można rozważać jako skrót notacyjny dla wyrażenia

$$(\texttt{fn } x \texttt{ => } e_2)\, e_1$$

(jest to tzw. *tłumaczenie Landina* [87]). Regułę typowania tego wyrażenia można więc również wyprowadzić z reguły typowania abstrakcji funkcyjnej.

Jeżeli w wyrażeniu

$$\texttt{let val rec } x = e_1 \texttt{ in } e_2 \texttt{ end}$$

wartość x jest zdefiniowana rekurencyjnie, to nie tylko wyrażenie e_2, lecz również e_1 może zawierać wolne wystąpienia zmiennej x. Aby więc dało się wyprowadzić typ wyrażenia e_1 należy uprzednio znać typ zmiennej x, ale jest on taki sam, jak wyrażenia e_1, jest ona bowiem jego nazwą. Zatem również w regule typowania pojawia się rekursja i pierwsza przesłanka tej reguły ma postać

$$\Gamma, x : \sigma \vdash e_1 : \sigma$$

(„dla e_1 można wyprowadzić typ σ przy założeniu, że zmienna x ma taki sam typ").

9.4.2. Wyprowadzanie typów a unifikacja

Zauważmy, że typy są termami nad jednogatunkową sygnaturą zawierającą stałe `int` i `bool` oraz dwa binarne symbole funkcyjne $\rightarrow$ i $*$, zmiennymi są zaś $\alpha, \beta, \ldots$. Reguły typowania zadają pewne zależności między typami wyrażeń i typami ich podwyrażeń. Załóżmy, że zamierzamy wyznaczyć typ pewnego wyrażenia e. Z każdym jego podwyrażeniem różnym od stałej i zmiennej (z każdym wierzchołkiem jego abstrakcyjnego drzewa rozbioru nie będącym liściem) związujemy zmienną typową α_i. Osobne zmienne typowe α_x, α_y, itd. związujemy z wszystkimi zmiennymi x, y itd. występującymi w programie. Ze stałymi zaś związujemy ich stałe typowe `int` i `bool`. Dla każdego podwyrażenia e_i wyrażenia e napiszemy równania postaci $\sigma \stackrel{?}{=} \tau$, gdzie σ i τ są typami, takie, że jeśli z wyrażeniami e_i, e_j i e_k związaliśmy zmienne typowe α_i, α_j i α_k oraz wyrażenie e_i jest postaci:

- $e_i = e_j + e_k$, to równaniami tymi są $\alpha_i \stackrel{?}{=} \texttt{int}$, $\alpha_j \stackrel{?}{=} \texttt{int}$ i $\alpha_k \stackrel{?}{=} \texttt{int}$;
- $e_i = (e_j, e_k)$, to równaniem tym jest $\alpha_i \stackrel{?}{=} \alpha_j * \alpha_k$;
- $e_i = (\texttt{fn } x \texttt{ => } e_j)$, to równaniem tym jest $\alpha_i \stackrel{?}{=} \alpha_x \rightarrow \alpha_j$ (α_x jest zmienną typową związaną ze zmienną x);
- $e_i = e_j e_k$, to równaniem tym jest $\alpha_j \stackrel{?}{=} \alpha_k \rightarrow \alpha_i$;
- $e_i = (\texttt{let val } x = e_j \texttt{ in } e_k \texttt{ end})$ lub $e_i = (\texttt{let val rec } x = e_j \texttt{ in } e_k \texttt{ end})$, to równaniami tymi są $\alpha_x \stackrel{?}{=} \alpha_j$ i $\alpha_i \stackrel{?}{=} \alpha_k$.

Twierdzenie 9.2. Opisany wyżej układ równań posiada unifikator θ (por. podrozdział 8.2) wtedy i tylko wtedy, gdy dla wyrażenia e można znaleźć pewien kontekst Γ i typ σ, takie, by $\Gamma \vdash e : \sigma$. Niech $x_1, \ldots, x_n$ będą zmiennymi wolnymi występującymi w wyrażeniu e. Wówczas jeśli istnieje unifikator θ opisanego wyżej układu, to

$$x_1 : \alpha_{x_1}\theta, \ldots, x_n : \alpha_{x_n}\theta \vdash e : \alpha_1\theta$$

gdzie α_{x_i} są zmiennymi typowymi związanymi ze zmiennymi x_i, zaś α_1 jest zmienną związaną z całym wyrażeniem e.

Prosty dowód tego twierdzenia można przeprowadzić przez indukcję względem struktury wyrażenia e_1. Wyprowadzanie typów w systemie monomorficznym sprowadziliśmy więc do zagadnienia unifikacji termów.

Ponieważ wśród unifikatorów układu równań istnieje unifikator najbardziej ogólny, również wśród tez typowych $\Gamma \vdash e : \sigma$, dotyczących ustalonego wyrażenia e, jeśli ich zbiór jest niepusty, istnieje teza najbardziej ogólna, zwana *tezą główną* (ang. *principal typing*). W szczególności dla wyrażenia e nie zawierającego zmiennych wolnych, jeśli posiada ono typ, to posiada ono *najbardziej ogólny typ* σ, zwany *typem głównym* (ang. *principal type*), taki, że jeśli $\vdash e : \tau$ dla pewnego typu τ, to istnieje podstawienie θ, takie że $\tau = \sigma\theta$. Najbardziej ogólny typ opisuje więc wszystkie możliwe typy wyrażenia. Na przykład typami wyrażenia `fn` x `=>` x są $\texttt{int} \to \texttt{int}$, $\texttt{bool} \to \texttt{bool}$, $\alpha \to \alpha$, $(\texttt{int} \to \alpha) \to (\texttt{int} \to \alpha)$ itd. Najogólniejszym typem jest $\alpha \to \alpha$. Każdy typ wyrażenia `fn` x `=>` x można otrzymać podstawiając w typie $\alpha \to \alpha$ pewien typ σ w miejsce zmiennej α.

Przykład 9.3. Wyprowadzimy typ główny wyrażenia

$$e_1 \quad = \quad (\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ (yx, x))(2 + 3)$$

Rozbijając wyrażenie e_1 na podwyrażenia mamy

$$\begin{aligned} e_1 &= e_2 e_3 \\ e_2 &= \texttt{fn}\ x\ \texttt{=>}\ e_4 \\ e_3 &= 2 + 3 \\ e_4 &= \texttt{fn}\ y\ \texttt{=>}\ e_5 \\ e_5 &= (e_6, x) \\ e_6 &= yx \end{aligned}$$

Z każdym podwyrażeniem e_i związujemy zmienną typową α_i, dla $i = 1, \ldots, 6$, ze zmienną x związujemy zmienną typową α_x, a ze zmienną y zmienną α_y. Zgodnie z opisanym wyżej algorytmem budowania układu równań mamy:

$$\begin{aligned} \alpha_2 &\stackrel{?}{=} \alpha_3 \to \alpha_1 \\ \alpha_2 &\stackrel{?}{=} \alpha_x \to \alpha_4 \\ \alpha_3 &\stackrel{?}{=} \texttt{int} \\ \texttt{int} &\stackrel{?}{=} \texttt{int} \\ \texttt{int} &\stackrel{?}{=} \texttt{int} \\ \alpha_4 &\stackrel{?}{=} \alpha_y \to \alpha_5 \\ \alpha_5 &\stackrel{?}{=} \alpha_6 * \alpha_x \\ \alpha_y &\stackrel{?}{=} \alpha_x \to \alpha_6 \end{aligned}$$

Najogólniejszym unifikatorem tego układu jest

$$\theta \ = \ [\ \alpha_1 \quad / \quad (\texttt{int} \to \alpha_6) \to (\alpha_6 * \texttt{int}),$$

$$\begin{array}{lcl} \alpha_2 & / & \texttt{int} \to (\texttt{int} \to \alpha_6) \to (\alpha_6\ \texttt{*int}), \\ \alpha_3 & / & \texttt{int}, \\ \alpha_4 & / & (\texttt{int} \to \alpha_6) \to (\alpha_6 * \texttt{int}), \\ \alpha_5 & / & \alpha_6 * \texttt{int}, \\ \alpha_x & / & \texttt{int}, \\ \alpha_y & / & \texttt{int} \to \alpha_6\ ] \end{array}$$

zatem

$$\begin{array}{lcl} e_1 & : & (\texttt{int} \to \alpha_6) \to (\alpha_6 * \texttt{int}) \\ e_2 & : & \texttt{int} \to (\texttt{int} \to \alpha_6) \to (\alpha_6\ \texttt{*int}) \\ e_3 & : & \texttt{int} \\ e_4 & : & (\texttt{int} \to \alpha_6) \to (\alpha_6 * \texttt{int}) \\ e_5 & : & \alpha_6 * \texttt{int} \\ e_6 & : & \alpha_6 \\ x & : & \texttt{int} \\ y & : & \texttt{int} \to \alpha_6 \end{array}$$

W szczególności typem głównym wyrażenia e_1 jest $(\texttt{int} \to \alpha) \to (\alpha * \texttt{int})$.

9.4.3. Algorytm rekonstrukcji typów w systemie monomorficznym

Algorytm opisany w poprzednim podrozdziale ma wielkie znaczenie teoretyczne — pokazuje, że zadanie rekonstrukcji typów jest nie trudniejsze od zadania unifikacji. Pozwala ponadto łatwo udowodnić wiele własności monomorficznego systemu typów, np. twierdzenie o istnieniu typu głównego. W praktycznych zastosowaniach byłby jednak nieporęczny, wymaga bowiem utworzenia układu równań o rozmiarze porównywalnym z rozmiarem całego programu. Do zastosowań praktycznych w kompilatorach wygodniejszy byłby algorytm, który analizuje program fragment po fragmencie, przechodząc przez jego abstrakcyjne drzewo rozbioru. W trakcie pracy takiego algorytmu gromadzi się pewna „wiedza" na temat typów zmiennych występujących w analizowanym programie. Przyjmijmy, że wiedza ta jest opisana za pomocą pewnego kontekstu Γ. Kontekst ten będzie zawierał wszystkie zmienne, które mają wolne wystąpienia w wyrażeniu e i tylko te. Algorytm rekonstrukcji typów jako dane wejściowe otrzymuje kontekst Γ i wyrażenie e, którego typ ma wyznaczyć. Ma rozwiązać zadanie: „wyznacz typ wyrażenia e przy założeniu, że typy występujących w nim zmiennych są ukonkretnieniami typów podanych w kontekście Γ". Wyznaczenie typu wyrażenia może pociągnąć za sobą ujawnienie dodatkowych zależności między typami, nie uwzględnionych w kontekście Γ. Dlatego wynikiem pracy algorytmu nie będzie tylko wyprowadzony typ wyrażenia, lecz para złożona z podstawienia i typu. Podstawienie to będzie ze sobą niosło tę dodatkową informację o zależnościach między typami. Algorytm opiszemy

podając zestaw równości postaci

$$W(\Gamma, e) \quad = \quad (\theta, \sigma)$$

Przykład 9.4. Niech $\Gamma = \{f : \texttt{bool} \to \texttt{int}, x : \alpha\}$ i $e = fx$. Typ wyrażenia e powinien być taki sam, jak typ wyniku funkcji f, narzuconego przez kontekst Γ, czyli `int`. Ponadto analiza wyrażenia e dostarcza dodatkowej informacji: typem zmiennej x musi być `bool`, żeby wyrażenie e miało typ. Zatem za zmienną typową α należy podstawić `bool` i ostatecznie

$$W(\{f : \texttt{bool} \to \texttt{int}, x : \alpha\},\ fx) \quad = \quad ([\alpha/\texttt{bool}], \texttt{int})$$

Opisywany algorytm jest przedstawiony w tablicy 9.5 na następnej stronie. Jest on zdefiniowany rekurencyjnie względem struktury analizowanego programu. Dla każdej produkcji gramatyki abstrakcyjnej języka (dla każdej reguły typowania) mamy odpowiednią regułę postaci $W(\Gamma, e) = (\theta, \sigma)$. Dla podstawienia θ i kontekstu $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ napis $\Gamma\theta$ oznacza kontekst $\{x_1 : \sigma_1\theta, \ldots, x_n : \sigma_n\theta\}$, zaś $\Gamma(x) = \sigma$, jeżeli $(x : \sigma) \in \Gamma$.

Jeżeli wyrażenie e jest stałą liczbową, to jego typem jest `int` w każdym kontekście Γ. Analiza tego wyrażenia nie dostarcza żadnych dodatkowych wiadomości na temat zależności pomiędzy typami występującymi w programie, dlatego wynikowym podstawieniem będzie podstawienie identycznościowe []. Podobnie jeżeli wyrażenie e jest stałą logiczną `true` lub `false`, wynikiem pracy algorytmu jest para złożona z podstawienia identycznościowego i typu `bool`. Jeżeli analizowane wyrażenie jest zmienną, to jego najbardziej ogólny typ jest typem, który zmiennej x narzuca kontekst Γ. Tutaj też nie otrzymujemy dodatkowej informacji o typach zmiennych, więc wynikowym podstawieniem będzie [].

Typem wyrażenia $e_1 + e_2$, jeśli posiada ono typ, jest `int`. Aby jednak stwierdzić, czy posiada ono typ i jakie zależności między typami implikuje fakt, że wyrażenia e_1 i e_2 są do siebie dodane musimy najpierw wyznaczyć typy wyrażeń e_1 i e_2. Następnie typy te powinno udać się zunifikować z typem `int`. Jeśli te trzy kroki przebiegną pomyślnie, otrzymujemy trzy podstawienia θ_1, θ_2 i θ_3, które złożone ze sobą dają podstawienie wynikowe.

Przykład 9.5. Niech $\Gamma = \{x : \alpha, y : \beta\}$. Rozważmy wyrażenie $e = x + y$. Mamy $W(\Gamma, x) = ([], \alpha)$ oraz $W(\Gamma[], y = ([], \beta)$. Zauważmy, że do typowania drugiego wyrażenia używamy kontekstu Γ na który nałożyliśmy podstawienie wyliczone w pierwszym kroku (w naszym przypadku identycznościowe). Ponieważ $\text{mgu}(\alpha \stackrel{?}{=} \texttt{int}, \beta \stackrel{?}{=} \texttt{int}) = [\alpha/\texttt{int}, \beta/\texttt{int}]$, to

$$W(\{x : \alpha, y : \beta\}, x + y) \quad = \quad ([\alpha/\texttt{int}, \beta/\texttt{int}], \texttt{int})$$

Aby wyznaczyć typ wyrażenia `fn` x `=>` e, musimy wpierw wyznaczyć typ wyrażenia e. Posiada ono jednak dodatkową zmienną wolną x. Do kontekstu Γ powinniśmy zatem dodać założenie o typie tej zmiennej. Ponieważ może on być dowolny, wybieramy nową zmienną typową β, nie występującą w kontekście Γ i do kontekstu Γ dodajemy założenie $x : \beta$. W tak otrzymanym kontekście wyznaczamy typ wyrażenia e. Wynikiem pracy algorytmu jest para złożona z pewnego podstawienia θ i typu σ. Podstawienie θ prawdopodobnie narzuca

$$W(\Gamma, c) = ([], \texttt{int}), \text{ jeśli } c \text{ jest stałą liczbową}$$

$$W(\Gamma, \texttt{true}) = ([], \texttt{bool})$$

$$W(\Gamma, \texttt{false}) = ([], \texttt{bool})$$

$$W(\Gamma, x) = ([], \Gamma(x))$$

$$\begin{aligned} W(\Gamma, e_1 + e_2) = \text{niech } & (\theta_1, \sigma_1) = W(\Gamma, e_1) \\ & (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\ & \theta_3 = \text{mgu}\{\sigma_1 \stackrel{?}{=} \texttt{int}, \sigma_2 \stackrel{?}{=} \texttt{int}\} \\ \text{jeżeli } & \theta_3 \text{ nie istnieje,} \\ & \text{to występuje błąd typowy,} \\ & \text{w p.p. wynikiem jest } (\theta_1\theta_2\theta_3, \texttt{int}) \end{aligned}$$

$$\begin{aligned} W(\Gamma, (e_1, e_2)) = \text{niech } & (\theta_1, \sigma_1) = W(\Gamma, e_1) \\ & (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\ \text{wynikiem jest } & (\theta_1\theta_2, \sigma_1\theta_2 * \sigma_2) \end{aligned}$$

$$\begin{aligned} W(\Gamma, \texttt{fn } x \texttt{ => } e) = \text{niech } & \beta = \text{nowa zmienna} \\ & (\theta, \sigma) = W(\Gamma \cup \{x : \beta\}, e) \\ \text{wynikiem jest } & (\theta, \beta\theta \to \sigma) \end{aligned}$$

$$\begin{aligned} W(\Gamma, e_1 e_2) = \text{niech } & \beta = \text{nowa zmienna} \\ & (\theta_1, \sigma_1) = W(\Gamma, e_1) \\ & (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\ & \theta_3 = \text{mgu}\{\sigma_1, \sigma_2 \to \beta\} \\ \text{jeżeli } & \theta_3 \text{ nie istnieje,} \\ & \text{to występuje błąd typowy,} \\ & \text{w p.p. wynikiem jest } (\theta_1\theta_2\theta_3, \beta\theta_3) \end{aligned}$$

$$\begin{aligned} W(\Gamma, \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) = \text{niech } & (\theta_1, \sigma_1) = W(\Gamma, e_1) \\ & (\theta_2, \sigma_2) = W(\Gamma\theta_1 \cup \{x : \sigma_1\}, e_2) \\ \text{wynikiem jest } & (\theta_1\theta_2, \sigma_2) \end{aligned}$$

$$\begin{aligned} W(\Gamma, \texttt{let val rec } x = e_1 \texttt{ in } e_2 \texttt{ end}) = \text{niech } & \beta = \text{nowa zmienna} \\ & (\theta_1, \sigma_1) = W(\Gamma \cup \{x : \beta\}, e_1) \\ & \theta_2 = \text{mgu}(\sigma_1, \beta\theta_1) \\ \text{jeżeli } & \theta_2 \text{ nie istnieje,} \\ & \text{to występuje błąd typowy, w p.p. niech} \\ & (\theta_3, \sigma_2) = W(\Gamma\theta_1\theta_2 \cup \{x : \sigma_1\theta_2\}, e_2) \\ \text{wynikiem jest } & (\theta_1\theta_2\theta_3, \sigma_2) \end{aligned}$$

Tablica 9.5. Algorytm rekonstrukcji typów w systemie monomorficznym

pewne ograniczenia na typ zmiennej x, jej typem jest więc $\beta\theta$. Typem wyrażenia `fn` x `=>` e jest zatem $\beta\theta \to \sigma$. Dla pozostałych konstrukcji składniowych języka Mini-ML algorytm działa podobnie.

Aby wyznaczyć tezę główną dla wyrażenia e, budujemy kontekst

$$\Gamma = \{x_1 : \alpha_1, \ldots, x_n : \alpha_n\}$$

gdzie α_i są parami różnymi zmiennymi typowymi, zawierający założenia typowe o wszystkich zmiennych mających wolne wystąpienia w wyrażeniu e i obliczamy $(\theta, \sigma) = W(\Gamma, e)$. Jeżeli algorytm zakończy się pomyślnie (nie przerwie obliczeń z komunikatem o błędzie), to teza $\Gamma\theta \vdash e : \sigma$ jest tezą główną dla wyrażenia e. W szczególności aby wyznaczyć typ główny programu e (program nie zawiera zmiennych wolnych) uruchamiamy algorytm dla pustego kontekstu $\Gamma = \emptyset$ i wyrażenia e. Otrzymujemy pewne podstawienie θ i typ σ. Typem głównym programu e jest σ.

9.4.4. System typowania z polimorfizmem parametrycznym

Idea polimorfizmu polega na założeniu, że różne wystąpienia tej samej zmiennej *w tym samym wyrażeniu* mogą mieć różne typy. W systemie monomorficznym funkcji `fn` x `=>` x możemy przypisać zarówno typ `int` $\to$ `int` jak i `bool` $\to$ `bool` a nawet jej typ główny $\alpha \to \alpha$. Jednak jeśli nadamy jej nazwę, np. f w wyrażeniu `let`, to każde wystąpienie zmiennej f w treści wyrażenia `let` musi mieć *taki sam* typ. Istotnie, typy zmiennych są jednoznacznie zadane przez kontekst. Dlatego wyrażenie

$$\texttt{let val } f = \texttt{fn } x \texttt{ => } x \texttt{ in } (f\ 1,\ f\ \texttt{true})\ \texttt{end}$$

nie posiada w systemie monomorficznym typu! Istotnie, wymaga on, by raz wyznaczony typ funkcji f był dobry tak dla wyrażenia f 1, jak i dla f `true`. Jeżeli dla f wyznaczymy typ `int` $\to$ `int`, wyrażenie f `true` będzie niepoprawne, podobnie jeśli dla f wyznaczymy typ `bool` $\to$ `bool`, wyrażenie f 1 będzie niepoprawne. Chcielibyśmy dla f wyznaczyć jej typ główny $\alpha \to \alpha$, jednak w kontekście $\{f : \alpha \to \alpha\}$ oba wyrażenia f 1 oraz f `true` są niepoprawne, wymagają one bowiem od f, by miała typ odpowiednio `int` $\to$ `int` i `bool` $\to$ `bool`. Wszystkiemu winien jest aksjomat

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

który wymusza, by każde wystąpienie zmiennej x miało *ten sam typ*, co typ wymieniony w kontekście. Wolelibyśmy, by typ zmiennej x po prawej stronie znaku $\vdash$ (typ jej ustalonego wystąpienia w wyrażeniu) był ukonkretnieniem typu występującego w kontekście, tak byśmy mogli napisać

$$f : \alpha \to \alpha \vdash f : \texttt{int} \to \texttt{int}$$

Aksjomat dla zmiennej mógłby mieć więc postać

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma\theta} \tag{9.1}$$

gdzie θ jest dowolnym podstawieniem. Korzystając z tego aksjomatu moglibyśmy napisać

$$f : \alpha \to \alpha \vdash f : (\alpha \to \alpha)[\alpha/\texttt{int}]$$

Przykład 9.6. Korzystając z aksjomatu (9.1) udowodnimy, że wyrażenie

$$\texttt{let val } f = \texttt{fn } x \texttt{ => } x \texttt{ in } (f\ 1,\ f\ \texttt{true})\ \texttt{end}$$

posiada typ `int * bool`. Istotnie, możemy przeprowadzić następujące wnioskowanie:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fn } x \texttt{ => } x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{}{f : \alpha \to \alpha \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{}{f : \alpha \to \alpha \vdash 1 : \texttt{int}}}{f : \alpha \to \alpha \vdash f\ 1 : \texttt{int}} \quad (*)}{f : \alpha \to \alpha \vdash (f\ 1,\ f\ \texttt{true}) : \texttt{int * bool}}}{\vdash \texttt{let val } f = \texttt{fn } x \texttt{ => } x \texttt{ in } (f\ 1,\ f\ \texttt{true})\ \texttt{end} : \texttt{int * bool}}$$

w którym dowód $(*)$ jest z braku miejsca przedstawiony poniżej:

$$\dfrac{\dfrac{}{f : \alpha \to \alpha \vdash f : \texttt{bool} \to \texttt{bool}} \quad \dfrac{}{f : \alpha \to \alpha \vdash \texttt{true} : \texttt{bool}}}{f : \alpha \to \alpha \vdash f\ \texttt{true} : \texttt{bool}}$$

Niestety aksjomat (9.1) może prowadzić do niedorzeczności. Kontekst $\{x : \alpha, y : \alpha\}$ mówi, że typy zmiennych x i y są takie same. Jeżeli skorzystamy z aksjomatu (9.1) i napiszemy

$$x : \alpha, y : \alpha \vdash x : \texttt{int}$$

to otrzymamy sprzeczność, gdyż w innym miejscu dowodu możemy wówczas napisać

$$x : \alpha, y : \alpha \vdash y : \texttt{bool}$$

Korzystając z reguły typowania pary możemy wówczas wywnioskować

$$x : \alpha, y : \alpha \vdash (x, y) : \texttt{int * bool}$$

co znaczy „w kontekście takim, że zmienne x i y mają ten sam typ, zmienna x ma typ `int`, zaś y typ `bool`". Nasz błąd polega na tym, że *w aksjomacie (9.1) po prawej stronie znaku* $\vdash$ *wolno podstawiać typy tylko za te zmienne, które nie występują w kontekście* Γ.

Ponadto *zmienne, w typach których wykonano jakiekolwiek podstawienia nie mogą być wiązane w abstrakcji funkcyjnej* (muszą być wiązane w wyrażeniu `let`). W przeciwnym razie system typów pozwoliłby na dodanie liczby do wartości logicznej:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : \alpha \to \texttt{int} \vdash f : \texttt{bool} \to \texttt{int}} \quad \dfrac{}{\ldots \vdash \texttt{true} : \texttt{bool}}}{f : \alpha \to \texttt{int} \vdash f\ \texttt{true} : \texttt{int}}}{\vdash \texttt{fn } f \texttt{ => } f\ \texttt{true} : (\alpha \to \texttt{int}) \to \texttt{int}} \qquad \dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \texttt{int}} \quad \dfrac{}{x : \alpha \vdash 1 : \texttt{int}}}{x : \alpha \vdash x + 1 : \texttt{int}}}{\vdash \texttt{fn } x \texttt{ => } x + 1 : \alpha \to \texttt{int}}}{\vdash (\texttt{fn } f \texttt{ => } f\ \texttt{true})\ (\texttt{fn } x \texttt{ => } x + 1) : \texttt{int}}$$

$$\frac{}{\Gamma \vdash c : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\sigma \vdash x : \sigma[\vec{\alpha}/\vec{\tau}]} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x\ \texttt{=>}\ e : \sigma \to \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \mathrm{Cl}_\Gamma(\sigma) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma\ \texttt{*}\ \tau} \qquad \frac{\Gamma, x : \sigma \vdash e_1 : \sigma \quad \Gamma, x : \mathrm{Cl}_\Gamma(\sigma) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val rec}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

Tablica 9.6. Polimorficzny system typów dla Mini-ML-a

Podstawiając w miejsce parametru f funkcję $\texttt{fn}\ x\ \texttt{=>}\ x + 1$ otrzymujemy wyrażenie

$$(\texttt{fn}\ x\ \texttt{=>}\ x + 1)\,\texttt{true}$$

które w wyniku podstawienia wartości `true` w miejsce parametru x upraszcza się do wyrażenia $\texttt{true} + 1$.

Wygodnie byłoby mieć możliwość oznaczenia w kontekście, za które zmienne typowe można podstawiać typy w aksjomacie (9.1). Dlatego wprowadzamy pojęcie *schematu typu* postaci $\forall\vec{\alpha}.\sigma$, w którym $\vec{\alpha}$ jest ciągiem zmiennych a σ typem i zmieniamy definicję kontekstu. Jest on od teraz zbiorem założeń typowych postaci $x : \forall\vec{\alpha}.\sigma$. Przyjmujemy ponadto, że jeśli ciąg zmiennych typowych $\vec{\alpha}$ jest pusty, to znak $\forall$ opuszczamy i taki schemat utożsamiamy z typem σ. Zatem w kontekście mogą występować zarówno typy (zwane dla podkreślenia faktu, że nie zawierają kwantyfikatora $\forall$ *typami otwartymi*), jak i schematy typów. Jednak po prawej stronie znaku $\vdash$ mogą występować jedynie typy. Założenie typowe postaci $x : \forall\vec{\alpha}.\sigma$ oznacza, że typem zmiennej x jest typ σ w którym za zmienne typowe $\vec{\alpha}$ można podstawić dowolne typy. Aksjomat (9.1) możemy więc teraz zapisać w postaci

$$\frac{}{\Gamma, x : \forall\vec{\alpha}.\sigma \vdash x : \sigma[\vec{\alpha}/\vec{\tau}]}$$

w której $[\vec{\alpha}/\vec{\tau}]$ oznacza podstawienie, którego nośnikiem jest zbiór zmiennych $\vec{\alpha}$. Tak zmodyfikowany system typów, zwany systemem typów z polimorfizmem parametrycznym lub systemem Damasa-Milnera jest przedstawiony w tablicy 9.6. W regułach typowania wyrażeń `let` napis $\mathrm{Cl}_\Gamma(\sigma)$, zwany *domknięciem typu*, oznacza schemat $\forall\vec{\alpha}.\sigma$, gdzie $\vec{\alpha}$ jest ciągiem wszystkich zmiennych, które występują w typie σ i nie występują w kontekście Γ. Dzięki temu możemy w pierwszej przesłance reguły `let` wyprowadzić typ główny σ dla wyrażenia e_1, zaś w jej drugiej przesłance możemy korzystać ze schematu typu $\forall\vec{\alpha}.\sigma$ dla zmiennej x, który pozwala na przypisywanie zmiennej x różnych konkretyzacji typu σ.

Przykład 9.7. Przepiszemy dowód z przykładu 9.6 w systemie z tablicy 9.6.

$$\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \texttt{fn}\ x\ \texttt{=>}\ x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{}{f : \forall\alpha.\alpha \to \alpha \vdash f : \texttt{int} \to \texttt{int}} \quad \dfrac{}{f : \forall\alpha.\alpha \to \alpha \vdash \texttt{1} : \texttt{int}}}{f : \forall\alpha.\alpha \to \alpha \vdash f\ \texttt{1} : \texttt{int}}\ (*)}{f : \forall\alpha.\alpha \to \alpha \vdash (f\ \texttt{1}, f\ \texttt{true}) : \texttt{int} * \texttt{bool}}}{\vdash \texttt{let val}\ f = \texttt{fn}\ x\ \texttt{=>}\ x\ \texttt{in}\ (f\ \texttt{1}, f\ \texttt{true})\ \texttt{end} : \texttt{int} * \texttt{bool}}$$

w którym dowód $(*)$ jest z braku miejsca przedstawiony poniżej:

$$\dfrac{\dfrac{}{f : \forall\alpha.\alpha \to \alpha \vdash f : \texttt{bool} \to \texttt{bool}} \quad \dfrac{}{f : \forall\alpha.\alpha \to \alpha \vdash \texttt{1} : \texttt{bool}}}{f : \forall\alpha.\alpha \to \alpha \vdash f\ \texttt{true} : \texttt{bool}}$$

9.4.5. Algorytm *W* Milnera

Mimo iż system monomorficzny i polimorficzny różnią się zasadniczo, algorytm rekonstrukcji typów opisany w podrozdziale 9.4.3 można łatwo przerobić tak, by rekonstruował typy w systemie polimorficznym. Teraz kontekst Γ będący parametrem wejściowym algorytmu będzie zawierał schematy typów. Algorytm ten jest przedstawiony w tablicy 9.7 na sąsiedniej stronie.

9.4.6. System z rekursją polimorficzną

Zauważmy, że mimo iż wprowadziliśmy schematy typów, to jednak w regule

$$\dfrac{\Gamma, x : \sigma \vdash e_1 : \sigma \quad \Gamma, x : \mathrm{Cl}_\Gamma(\sigma) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val rec}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

wyrażenie e_1 jest typowane monomorficznie. Jego typ jest przekształcony do schematu typu dopiero w drugiej przesłance. Moglibyśmy ulec pokusie „poprawienia” tej reguły w następujący sposób:

$$\dfrac{\Gamma, x : \mathrm{Cl}_\Gamma(\sigma) \vdash e_1 : \sigma \quad \Gamma, x : \mathrm{Cl}_\Gamma(\sigma) \vdash e_2 : \tau}{\Gamma \vdash \texttt{let val rec}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end} : \tau}$$

System typów z taką regułą został zdefiniowany przez A. Mycrofta [120], dlatego bywa nazywany systemem Mycrofta-Milnera. Jego wadą jest to, że nie istnieje dla niego algorytm rekonstrukcji typów. Mimo to jest on wykorzystywany w niektórych językach programowania, w szczególności istnieją rozszerzenia kompilatora SML/NJ wykorzystujące ten system typów. System Mycrofta-Milnera jest konserwatywnym rozszerzeniem systemu Damasa-Milnera, tj. wszystkie programy, które posiadają typ w systemie Damasa-Milnera mają również typ w systemie Mycrofta-Milnera (niekiedy bardziej ogólny), jednak nie odwrotnie. Przykładem programu w Mini-ML-u, który nie posiada typu w systemie Damasa-Milnera a ma taki typ w systemie Mycrofta-Milnera jest

$$\texttt{fn}\ g\ \texttt{=>}\ \texttt{let val rec}\ f = \texttt{fn}\ x\ \texttt{=>}\ g\,(f\ \texttt{1})\,(f\ \texttt{true})\ \texttt{in}\ f\ f\ \texttt{end}$$

$$
\begin{aligned}
W(\Gamma, c) &= ([], \texttt{int}), \text{ jeśli } c \text{ jest stałą liczbową} \\
W(\Gamma, \texttt{true}) &= ([], \texttt{bool}) \\
W(\Gamma, \texttt{false}) &= ([], \texttt{bool}) \\
W(\Gamma, x) &= \text{niech } \vec{\beta} = \text{ ciąg nowych zmiennych} \\
&\qquad \forall \vec{\alpha}.\sigma = \Gamma(x) \\
&\quad \text{wynikiem jest } ([], \sigma[\vec{\alpha}/\vec{\beta}]) \\
W(\Gamma, e_1 + e_2) &= \text{niech } (\theta_1, \sigma_1) = W(\Gamma, e_1) \\
&\qquad (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\
&\qquad \theta_3 = \text{mgu}\,\{\sigma_1 \stackrel{?}{=} \texttt{int}, \sigma_2 \stackrel{?}{=} \texttt{int}\} \\
&\quad \text{jeżeli } \theta_3 \text{ nie istnieje,} \\
&\qquad \text{to występuje błąd typowy,} \\
&\qquad \text{w p.p. wynikiem jest } (\theta_1\theta_2\theta_3, \texttt{int}) \\
W(\Gamma, (e_1, e_2)) &= \text{niech } (\theta_1, \sigma_1) = W(\Gamma, e_1) \\
&\qquad (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\
&\quad \text{wynikiem jest } (\theta_1\theta_2, \sigma_1\theta_2 \texttt{ * } \sigma_2) \\
W(\Gamma, \texttt{fn } x \texttt{ => } e) &= \text{niech } \beta = \text{nowa zmienna} \\
&\qquad (\theta, \sigma) = W(\Gamma \cup \{x : \beta\}, e) \\
&\quad \text{wynikiem jest } (\theta, \beta\theta \to \sigma) \\
W(\Gamma, e_1 e_2) &= \text{niech } \beta = \text{nowa zmienna} \\
&\qquad (\theta_1, \sigma_1) = W(\Gamma, e_1) \\
&\qquad (\theta_2, \sigma_2) = W(\Gamma\theta_1, e_2) \\
&\qquad \theta_3 = \text{mgu}\,\{\sigma_1, \sigma_2 \to \beta\} \\
&\quad \text{jeżeli } \theta_3 \text{ nie istnieje,} \\
&\qquad \text{to występuje błąd typowy,} \\
&\qquad \text{w p.p. wynikiem jest } (\theta_1\theta_2\theta_3, \beta\theta_3) \\
W(\Gamma, \texttt{let val } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= \text{niech } (\theta_1, \sigma_1) = W(\Gamma, e_1) \\
&\qquad \Gamma_1 = \Gamma\theta_1 \\
&\qquad (\theta_2, \sigma_2) = W(\Gamma_1 \cup \{x : \text{Cl}_{\Gamma_1}(\sigma_1)\}, e_2) \\
&\quad \text{wynikiem jest } (\theta_1\theta_2, \sigma_2) \\
W(\Gamma, \texttt{let val rec } x = e_1 \texttt{ in } e_2 \texttt{ end}) &= \text{niech } \beta = \text{nowa zmienna} \\
&\qquad (\theta_1, \sigma_1) = W(\Gamma \cup \{x : \beta\}, e_1) \\
&\qquad \theta_2 = \text{mgu}(\sigma_1, \beta\theta_1) \\
&\quad \text{jeżeli } \theta_2 \text{ nie istnieje,} \\
&\qquad \text{to występuje błąd typowy, w p.p. niech} \\
&\qquad \Gamma_1 = \Gamma\theta_1\theta_2 \\
&\qquad (\theta_3, \sigma_2) = W(\Gamma_1 \cup \{x : \text{Cl}_{\Gamma_1}(\sigma_1\theta_2)\}, e_2) \\
&\quad \text{wynikiem jest } (\theta_1\theta_2\theta_3, \sigma_2)
\end{aligned}
$$

Tablica 9.7. Algorytm W Milnera rekonstrukcji typów

9.5. Zadania

Zadanie 9.1. Oto fragment deklaracji w C++:

```
const int DIM=3;
typedef float vector[DIM];
```

Zdefiniuj w C++ przeciążony operator `*` dla skalarnego mnożenia elementów typu `vector`.

Zadanie 9.2. Używając szablonów w C++ napisz polimorficzną funkcję sortującą elementy dowolnego typu `Elem` tablicy rozmiaru `N` zgodnie z podaną funkcją typu `int (Elem,Elem)` (stała `N` i typ `Elem` powinny być parametrami wzorca). Zaimplementuj przy tym algorytm sortowania przez proste wybieranie.

Zadanie 9.3. Podaj po kilka przykładów deklaracji w SML-u wartości typu:

1. `int`
2. `int -> int -> int`
3. `int * int -> int`
4. `int -> int * int`
5. `{re : real, im : real}`
6. `{a : int, b : int, c : int}`
7. `unit`
8. `'a -> unit`
9. `unit -> unit`
10. `unit -> string`
11. `'a -> int * string`
12. `'a -> string -> bool`
13. `'a -> 'a`
14. `'a -> 'b`
15. `'a -> 'b -> 'a`
16. `''a -> ''a`
17. `'a -> 'a list * 'a`
18. `'a -> 'b list`
19. `('a->'b) -> 'a list->'b list`
20. `('a->'b) -> 'b list->'a list`

Zadanie 9.4. Wskaż, które z poniższych deklaracji w SML-u są poprawne pod względem typów i wyznacz najogólniejsze typy deklarowanych funkcji. Uzasadnij dlaczego pozostałe deklaracje są niepoprawne.

1. `fun f x y = x () y`
2. `fun f x y = y x ()`
3. `fun f x y = (x,y,x)`
4. `fun f x y = ((x,y),x)`
5. `fun f x y = (x,(y,x))`
6. `fun f x y = (x y,x())`
7. `fun f x y = (x y,y())`
8. `fun f x y = (x y,y x)`
9. `fun f x y = (x y,x y y)`
10. `fun f x y = (x y,y x x)`
11. `fun f x y = x(y,y)`
12. `fun f x y = x(y,y x)`
13. `fun f x y = (x=y)`
14. `fun f x y = (x=y,y=x)`
15. `fun f x y = {x=y,y=x}`
16. `fun f x y = [x=y,y=x]`

17. `fun f x y = (x mod y,y mod x)`
18. `fun f x y = x+y+1`
19. `fun f x y = x*y`
20. `fun f x y = x/y`
21. `fun f x y = ([x],y)`
22. `fun f x y = (x, [y], [[x]])`
23. `fun f x y = ([x]::y, [[y]])`
24. `fun f x y = ([x]::[y], [[y]])`
25. `fun f x y = [y]@x`
26. `fun f x y = x::x`
27. `fun f x y = x::[x]`
28. `fun f x y = x::x::y`
29. `fun f x y = x::x@y`
30. `fun f x y = x::y@y`
31. `fun f x y = x@x::y`
32. `fun f x y = x@y::x`
33. `fun f x y = x^x^y`
34. `fun f x y = x(y^x)`
35. `fun f x y = x y^y`
36. `fun f x y = x^y::y`
37. `fun f x y = x = x = y`
38. `fun f x y = x = y = x`
39. `fun f x y = x y = 0`
40. `fun f x y = x (y = 0)`

Zadanie 9.5. Napisz dowód sądu typowego

$$\vdash \texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ yxx\ :\ \texttt{int} \to (\texttt{int} \to \texttt{int} \to \texttt{bool}) \to \texttt{bool}$$

Zadanie 9.6. Wskaż, które wyrażenia w Mini-ML-u posiadają typ w monomorficznym systemie typów. Wyznacz dla nich typy główne i napisz dowody używając reguł typowania. Uzasadnij dlaczego pozostałe wyrażenia nie mają typu.

1. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x$
2. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y$
3. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ xx$
4. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ yx$
5. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ xyx$
6. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ yxx$
7. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yx)$
8. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(yx)$
9. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ yxxx$
10. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(xy)x$
11. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(yx)x$
12. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ xy(yx)$
13. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ yx(yx)$
14. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yxx)$
15. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(yxx)$
16. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(x(yx))$
17. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(y(yx))$
18. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(y(xx))$
19. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(y(yx))$
20. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ xyyyy$
21. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(xy)(xy)$
22. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(xy)(yx)$
23. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yx)(xy)$
24. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yx)(yx)$
25. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yx(xy))$
26. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yx(yx))$
27. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(yxxx)$
28. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ y(yxxx)$
29. $\texttt{fn}\ x\ \texttt{=>}\ \texttt{fn}\ y\ \texttt{=>}\ x(y(xy)x)$

30. `fn` x `=>` `fn` y `=>` $x(y(yx)x)$
31. `fn` x `=>` `fn` y `=>` $x(x(xyy))$
32. `fn` x `=>` `fn` y `=>` $x(x(yxx))$
33. `fn` x `=>` `fn` y `=>` $x(y(yxx))$
34. `fn` x `=>` `fn` y `=>` $x(x(x(xy)))$
35. `fn` x `=>` `fn` y `=>` $x(x(x(yx)))$
36. `fn` x `=>` `fn` y `=>` $x(x(y(xy)))$
37. `fn` x `=>` `fn` y `=>` $xy(yxx)$
38. `fn` x `=>` `fn` y `=>` $xy(x(xy))$
39. `fn` x `=>` `fn` y `=>` $xy(x(yx))$
40. `fn` x `=>` `fn` y `=>` $yx(x(yx))$
41. `fn` x `=>` $x(x($`fn` y `=>` $x))$
42. `fn` x `=>` $x(x($`fn` x `=>` $x))$
43. `fn` x `=>` $x($`fn` x `=>` `fn` x `=>` $x)$
44. `fn` x `=>` $x(x($`fn` x `=>` `fn` x `=>` $x))$
45. `fn` x `=>` $x($`fn` y `=>` $x($`fn` x `=>` $x))$
46. `fn` x `=>` $x($`fn` y `=>` $yxx)$
47. `fn` x `=>` $x($`fn` y `=>` $y($`fn` y `=>` $x))$
48. `fn` x `=>` $x($`fn` x `=>` $x($`fn` x `=>` $x))$
49. `fn` x `=>` $x($`fn` x `=>` $($`fn` x `=>` $x)x)$
50. `fn` x `=>` $x($`fn` x `=>` `fn` x `=>` $x)$

Zadanie 9.7. Wyprowadź w systemie Damasa-Milnera najogólniejszy typ wyrażenia

$$\texttt{let}\ i = \texttt{fn}\ x \texttt{ => } x\ \texttt{in}\ ii\ \texttt{end}$$

Zadanie 9.8. Rozważmy system, w którym konstrukcja $\forall\alpha.\sigma$ nie jest schematem typu, lecz zwykłym typem, tj. język, którego abstrakcyjna składnia jest następująca:

$$\begin{array}{rcl} e & ::= & x \mid e_1 e_2 \mid \texttt{fn}\ x \texttt{ => } e \\ \sigma & ::= & \alpha \mid \sigma_1 \to \sigma_2 \mid \forall\alpha.\sigma \end{array}$$

i w którym przyjęto następujące reguły typowania:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x \texttt{ => } e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash e : \forall\alpha.\sigma}\ \alpha \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha/\tau]}$$

Jest to tak zwany rachunek typów drugiego rzędu. Pokaż, że w tym systemie wyrażenie `fn` x `=>` xx posiada typ, podczas gdy w języku z tzw. *płytkim polimorfizmem*, tj. w systemie Damasa-Milnera — nie.

Zadanie 9.9. Oto abstrakcyjna składnia pewnego języka:

$$\begin{array}{rcl} e & ::= & x \mid e_1 e_2 \mid \texttt{fn}\ x \texttt{ => } e \\ \sigma & ::= & \alpha \mid \sigma_1 \to \sigma_2 \end{array}$$

Reguły typowania są monomorficzne:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x \texttt{ => } e : \sigma \to \tau}$$

Rozmiar wyrażenia i typu definiujemy podobnie jak w zadaniu 8.2. Pokaż, że istnieje wyrażenie rozmiaru n, którego typ ma rozmiar $2^{\Omega(n)}$ (wzoruj się na podobnym przykładzie dla zadania unifikacji).

Zadanie 9.10. Pokaż, że w Mini-ML-u z polimorficznym systemem typów Damasa-Milnera istnieje wyrażenie rozmiaru n, którego typ ma rozmiar $2^{2^{\Omega(n)}}$. Opisz rolę polimorfizmu w budowaniu tak wielkiego typu. Pokaż, że w Mini-ML-u z monomorficznym systemem typów typ wyrażenia rozmiaru n ma rozmiar $2^{O(n)}$.

Zadanie 9.11. Oto abstrakcyjna składnia pewnego języka:

$$\begin{aligned} e &::= x \mid e_1 e_2 \mid \texttt{fn}\ x : \sigma\ \texttt{=>}\ e \\ \sigma &::= o \mid \sigma_1 \to \sigma_2 \end{aligned}$$

gdzie o jest stałą (o oznacza typ *bazowy*, taki jak `int`). Nie ma zmiennych typowych. Jest to zatem język, w którym typy są monomorficzne i występują *explicite* (tak jak w Pascalu). Reguły typowania są następujące:

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{fn}\ x : \sigma\ \texttt{=>}\ e : \sigma \to \tau}$$

Wyrażenie jest *zamknięte*, jeśli nie zawiera zmiennych wolnych. Wprowadzamy pojęcie *rzędu* typu:

$$\begin{aligned} \mathrm{order}(o) &= 1 \\ \mathrm{order}(\sigma_1 \to \ldots \to \sigma_n \to o) &= 1 + \max(\mathrm{order}(\sigma_1), \ldots, \mathrm{order}(\sigma_n)) \end{aligned}$$

Moc typu, oznaczana $\mathrm{card}(\sigma)$, to liczba różnych zamkniętych zredukowanych wyrażeń tego typu (utożsamiamy wyrażenia różniące się jedynie nazwami zmiennych związanych). Napis $\sigma^k \to \tau$ oznacza $\sigma \to \ldots \to \sigma \to \tau$, gdzie σ występuje k razy. Formalnie $\sigma^0 \to \tau = \tau$, $\sigma^{k+1} \to \tau = \sigma \to \sigma^k \to \tau$. Udowodnij twierdzenie:

1. Jeśli $\mathrm{order}(\sigma) = 1$, to $\sigma = o$ i $\mathrm{card}(\sigma) = 0$.
2. Jeśli $\mathrm{order}(\sigma) = 2$, to $\sigma = o^k \to o$ i $\mathrm{card}(o^k \to o) = k$.
3. Jeśli $\mathrm{order}(\sigma) > 2$, to $\mathrm{card}(\sigma) = 0$ lub $\mathrm{card}(\sigma) = \infty$. Ponadto $\mathrm{card}(\sigma \to \tau) = \infty$ wtedy i tylko wtedy, gdy $\mathrm{card}(\sigma) = 0$ lub $\mathrm{card}(\tau) = \infty$.

Wniosek: dla dowolnego typu σ mamy: $\mathrm{card}(\sigma) > 0$ wtedy i tylko wtedy, gdy σ przeczytane jako formuła klasycznego rachunku zdań, w której o oznacza fałsz, zaś $\to$ implikację jest prawdziwa. Typy możemy więc utożsamiać z formułami logicznymi a wyrażenia tych typów z ich dowodami. Np. wyrażenie `fn` $x : o$ `=>` x jest dowodem formuły $o \to o$. Nie ma wyrażeń typu $(o \to o) \to o$ (bo ta formuła nie jest prawdziwa, więc nie ma dowodu). Jest to tzw. *izomorfizm Curry'ego-Howarda*.

Zadanie 9.12. Typ w Mini-ML-u z monomorficznym systemem typów jest *niepusty*, jeśli istnieje zamknięte wyrażenie tego typu. Wskaż typy niepuste. Dla każdego z nich podaj przykład zamkniętego wyrażenia, dla którego jest on typem głównym. Dla pozostałych typów uzasadnij, dlaczego są puste.

1. $\alpha \to \alpha$
2. $\alpha \to \alpha \to \alpha$
3. $\alpha \to \beta \to \alpha$
4. $\alpha \to \beta \to \beta$
5. $(\alpha \to \beta) \to \alpha$
6. $\alpha \to \alpha \to \alpha \to \alpha$
7. $\alpha \to \alpha \to \alpha \to \beta$
8. $\alpha \to \alpha \to \beta \to \alpha$
9. $\alpha \to (\alpha \to \alpha) \to \alpha$
10. $\alpha \to (\alpha \to \beta) \to \alpha$
11. $\alpha \to (\alpha \to \beta) \to \beta$
12. $\alpha \to (\beta \to \alpha) \to \alpha$
13. $\alpha \to (\beta \to \alpha) \to \beta$
14. $(\alpha \to \alpha) \to \beta \to \beta$
15. $(\alpha \to \beta) \to \beta \to \alpha$
16. $(\alpha \to \alpha \to \alpha) \to \alpha$
17. $((\alpha \to \alpha) \to \alpha) \to \alpha$
18. $((\alpha \to \alpha) \to \beta) \to \beta$
19. $((\alpha \to \beta) \to \alpha) \to \alpha$
20. $((\alpha \to \beta) \to \beta) \to \alpha$

Rozdział 10

Esej o ciągu Fibonacciego

Pomyśl dwa razy nim zaczniesz programować,
byś nie musiał programować dwa razy nim zaczniesz myśleć.

Ciąg Fibonacciego jest ciągiem dodatnich liczb całkowitych zadanym następującą zależnością rekurencyjną:

$$\begin{aligned} F_0 &= 1 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \qquad n \geq 2 \end{aligned}$$

Na przykładzie problemu wyznaczenia wartości n-tego wyrazu ciągu Fibonacciego opiszemy ideę *polimorfizmu* i *powtórnego użycia kodu* (ang. *code reuse*) w programowaniu.

Naiwność. Narzuca się oczywista, aczkolwiek bardzo mało efektywna implementacja ciągu F_n w SML-u:

```
fun fib1 0 = 1
  | fib1 1 = 1
  | fib1 n = fib1 (n-1) + fib1 (n-2)
```

Fakt 10.1. Dla dowolnego $n \geq 0$

$$F_n \geq \left(\frac{3}{2}\right)^n$$

Fakt 10.2. Liczba wywołań rekurencyjnych funkcji `fib1` wynosi nie mniej niż F_n dla $n \geq 0$.

Liczba wywołań rekurencyjnych funkcji `fib1` rośnie zatem wykładniczo względem n i taka implementacja jest w praktyce bardzo nieefektywna (por. tablica 10.1).

n	czas działania [s]	
	`fib1`	`fib2`
20	0.01	0.00
25	0.12	0.00
30	1.43	0.00
35	15.30	0.00
40	174.00	0.00
43	740.86	0.00
44	`Overflow`	

Tablica 10.1. Porównanie czasów działania funkcji `fib1` i `fib2` (SPARCstation 4 z 1995 roku, zegar 110 MHz)

Roztropność. Stosując znane rozwiązanie (nazywane niekiedy *memoizacją*[1]), polegające na zapamiętywaniu wyników dwóch kolejnych wartości ciągu F_n, możemy zmniejszyć liczbę iteracji niezbędnych do wyznaczenia wartości F_n z $\Omega(2^n)$ do n:

```
local
   fun aux (a,b) n =
          if n <= 1
             then b
             else aux (b,a+b) (n-1)
in
   val fib2 = aux (1,1)
end
```

Funkcje `fib1` i `fib2` różnie zachowują się dla argumentów spoza dziedziny funkcji F_n, tj. dla liczb ujemnych (jedna się zapętla, co może ułatwić znalezienie błędu w programie korzystającym z tej funkcji, druga zwraca wartość `1`, co można uznać za rozszerzenie definicji funkcji F_n na liczby ujemne). Z punktu widzenia niezawodności oprogramowania, obie wersje są niezadowalające. W SML-u możemy takie sytuacje rozwiązywać bardzo elegancko używając wyjątków. Dla prostoty pozostaniemy jednak przy tych mniej eleganckich rozwiązaniach.

Porównanie czasów działania obu algorytmów, przedstawione w tablicy 10.1, dowodzi, że funkcja `fib2` w praktyce sprawdza się bardzo dobrze. Nie zatem względy praktyczne, tylko chęć dotarcia do prawdy zmusza nas do poszukiwania jeszcze sprawniejszego algorytmu. Faktycznie, istnieje algorytm wykonujący jedynie $\log(n)$ iteracji.

[1] Memoizacja to technika programowania i/lub kompilacji programu polegająca na przechowywaniu wyników wywołań funkcji. Gdy następuje kolejne wywołanie funkcji dla argumentu, dla którego była już ona obliczana wcześniej, to nie jest ona powtórnie wykonywana, zaś jej wynik jest natychmiast odczytywany z pamięci. Memoizacja może być stosowana jedynie do funkcji, które nie powodują skutków ubocznych (dlaczego?).

Spryt. Rozważmy macierz:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

Fakt 10.3. Niech

$$A^n = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

dla $n \geq 0$. Wówczas:

$$b_{22} = F_n$$

Dowód: indukcyjny względem n.

Zauważmy, że n-tą potęgę macierzy potrafimy wyliczyć wykonując $\log n$ iteracji za pomocą wielokrotnego podnoszenia do kwadratu:

$$\begin{aligned} A^0 &= I \\ A^{2k} &= \left(A^k\right)^2 \\ A^{2k+1} &= \left(A^k\right)^2 \times A \end{aligned}$$

gdzie

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

jest macierzą jednostkową. Pozostaje zaprogramować odpowiednie funkcje w SML-u.

Macierze 2 × 2. Macierze całkowitoliczbowe rozmiaru 2 × 2 reprezentujemy jako pary wierszy będących parami liczb całkowitych[2] (są więc typu `(int*int)*(int*int)`). Na pewno przyda się macierz jednostkowa. Infiksowy operator `**` mnożenia macierzy implementujemy wprost z definicji:

```
val I = ((1,0),(0,1))
infix 7 **
fun ((a11,a12),(a21,a22)) ** ((b11,b12),(b21,b22)) =
      ((a11*b11+a12*b21, a11*b12+a12*b22),
       (a21*b11+a22*b21, a21*b12+a22*b22))
```

[2] „Niektórzy programują rzeczy bardziej zajmujące, niż mnożenie macierzy rozmiaru 2 × 2.” (Oskar Miś)

Potęgowanie. Powyżej zdefiniowaliśmy półgrupę `((int*int)*(int*int), **, I)` macierzy całkowitoliczbowych rozmiaru 2×2 z mnożeniem `**` i elementem neutralnym `I`.

Operacja podnoszenia do naturalnej potęgi jest możliwa do zdefiniowania nie tylko w świecie macierzy rozmiaru 2×2, ale w dowolnej półgrupie $(M, \otimes, e)$ z binarnym operatorem $\otimes$ i elementem neutralnym e:

$$\begin{aligned} x^0 &= e \\ x^n &= x^{n-1} \otimes x \end{aligned}$$

dla dowolnego $x \in M$.

Dla przykładu tą półgrupą może być zbiór liczb całkowitych lub rzeczywistych z mnożeniem, albo zbiór liczb całkowitych modulo n z mnożeniem. Pierścień liczb całkowitych modulo n:

$$\mathbb{Z}_n = (\{0, \ldots, n-1\}, \texttt{addMod}\ n, \texttt{multMod}\ n, 0, 1)$$

implementujemy w SML-u definiując operacje dodawania i mnożenia modulo n:

```
local
   fun opMod op+ n (x,y) = (x+y) mod n
in
   val addMod  = opMod op+
   val multMod = opMod op*
end
```

Liczbę iteracji niezbędnych do wyliczenia x^n możemy zmniejszyć z n do $\log n$ za pomocą wielokrotnego podnoszenia do kwadratu (podobnie jak dla macierzy):

$$\begin{aligned} x^0 &= e \\ x^{2k} &= x^k \otimes x^k \\ x^{2k+1} &= x^k \otimes x^k \otimes x \end{aligned}$$

Powyższe równości są słuszne pod warunkiem, że $(M, \otimes, e)$ jest półgrupą z jednością (działanie $\otimes$ powinno być łączne, zaś e powinien być elementem neutralnym).

Zamiast więc programować konkretną funkcję potęgującą macierze, definiujemy polimorficzną funkcję `power`:

```
fun power (_,e) (_,0) = e
  | power (semigroup as (op*,_)) (x,k) =
       let
          val y = power semigroup (x, k div 2)
       in
          if k mod 2 = 0
             then y * y
             else y * y * x
       end
```

w której `semigroup as (op*,e)` jest parametrem. Dzięki temu algorytm potęgowania jest bardziej przejrzysty, nie zawiera nieistotnych detali związanych z macierzami. Konkretny operator `powmtx` potęgowania macierzy otrzymujemy już natychmiast. Gratis dostajemy także funkcje `powi`, `powr` i `powMod` potęgowania liczb całkowitych, rzeczywistych i całkowitych modulo *n*, podając jako parametr funkcji `power` definicje odpowiednich półgrup:

```
infix 8 powmtx powi powr
val op powmtx = power (op**,I)
val op powi   = power (op*,1)
val op powr   = power (op*,1.0)
fun powMod n  = power (multMod n, 1)
```

Dla powyższych operatorów wybrałem łączność w lewo (czemu, skoro operator potęgowania wiąże zwykle w prawo?). Jeżeli zamierzamy zmieniać łączliwość identyfikatora, lepiej zrobić to *przed* związaniem z nim wartości, nie *po* (czemu?).
Na koniec pozostaje zdefiniować trzecią wersję funkcji Fibonacciego:

```
local
   val A = ((0,1),(1,1))
in
   fun fib3 n = (#2 o #2) (A powmtx n)
end
```

Czas działania funkcji `fib3`, podobnie jak `fib2`, jest poniżej możliwości pomiarowych.

Mania wielkości. Ponieważ zakres liczb całkowitych jest ograniczony, a wartości ciągu Fibonacciego rosną wykładniczo i już dla $n = 44$ występuje nadmiar,[3] warto zaprogramować algorytm obliczający zmiennoprzecinkowe przybliżenie ciągu F_n. Można by przepisać definicję np. funkcji `fib1` zamieniając wartości całkowitoliczbowe na rzeczywiste:

```
fun fib1r 0 = 1.0
  | fib1r 1 = 1.0
  | fib1r n = fib1r (n-1) + fib1r (n-2)
```

Tak uczyniłby jedynie początkujący programista! Doświadczony programista przystępując do programowania ciągu Fibonacciego pyta się, *co jest istotnego w jego definicji*, podobnie jak postąpiliśmy programując operację potęgowania. Ciąg Fibonacciego można zdefiniować nie tylko dla liczb całkowitych, ale w dowolnej półgrupie $(G, \oplus)$ przez wskazanie dwóch pierwszych wyrazów ciągu $a, b \in G$:

$$\begin{aligned} F_0 &= a \\ F_1 &= b \\ F_n &= F_{n-1} \oplus F_{n-2} \end{aligned}$$

[3] A więc o to chodziło Mickiewiczowi!

Reszta to nieistotne detale. Wykorzystując polimorfizm możemy zdefiniować funkcję `fib` (według algorytmu wykorzystanego w funkcji `fib2`) podając jej operację $\oplus$ i wartości początkowe (a, b) jako parametry:

```
fun fib  _  (a,_) 0 = a
  | fib  _  (_,b) 1 = b
  | fib op+ (a,b) n = fib op+ (b,a+b) (n-1)
```

Deklaracja funkcji `fibi` dla liczb całkowitych, implementującej dokładnie ten sam algorytm, co funkcja `fib2`, mieści się teraz w jednej linijce. Zmiennoprzecinkowa wersja tej funkcji zajmuje kolejną linijkę. Nadto gratis dostajemy funkcję obliczającą tzw. słowa Fibonacciego nad alfabetem {`a`, `b`}:

```
val fibi = fib op+ (1,1)
val fibr = fib op+ (1.0,1.0)
val fibs = fib op^ ("a","b")
```

Przezorność. Zmiennoprzecinkowe przybliżenia liczb Fibonacciego możemy obliczać z jawnego wzoru na n-ty wyraz ciągu, otrzymanego przez rozwiązanie zależności rekurencyjnej metodą podstawiania. Załóżmy że $F_n = r^n$. Skoro

$$F_n \quad = \quad F_{n-1} + F_{n-2} \tag{10.1}$$

to $r^2 - r - 1 = 0$, czyli

$$r_1 = \frac{1+\sqrt{5}}{2} \qquad r_2 = \frac{1-\sqrt{5}}{2}$$

Równanie (10.1) jest liniowe, więc dowolna kombinacja liniowa wartości r_1 i r_2 jest również rozwiązaniem równania (10.1). Zatem

$$F_n = c_1 r_1^n + c_2 r_2^n$$

Stałe c_1 i c_2 wyznaczamy z warunków początkowych $F_0 = 1$ i $F_1 = 1$:

$$c_1 = \frac{1+\sqrt{5}}{2\sqrt{5}} \qquad c_2 = 1 - c_1$$

Ostatecznie:

```
local
   val sqrt5 = Math.sqrt 5.0
   val c1 = (1.0 + sqrt5) / (2.0 * sqrt5)
   val c2 = 1.0 - c1
   val r1 = 0.5 * (1.0 + sqrt5)
   val r2 = 0.5 * (1.0 - sqrt5)
in
   fun fibre n = c1 * r1 powr n + c2 * r2 powr n
end
```

Funkcja `powr` (a wydawała się niepotrzebna, gdy zajmowaliśmy się potęgowaniem!) tu się przydaje i nie musimy jej ponownie programować.

Szybkość. Do eksperymentów można wykorzystać funkcję `time` dokonującą pomiaru czasu działania procesora (moduły `Time` i `Timer` są częścią biblioteki standardowej SML-a):

```
fun time (f : int -> int) (x : int) : unit =
   let
      val timer = Timer.startCPUTimer ()
      val result = f x
      val {usr, sys, gc} = Timer.checkCPUTimer timer
   in
      print ("arg=" ^ Int.toString x ^
             ", result=" ^ Int.toString result ^
             ", time=" ^   Time.toString usr ^
             ", gc=" ^     Time.toString gc ^
             ", system=" ^ Time.toString sys ^
             "\n")
   end
```

Dla przykładu:

```
- time fib1 30;
arg=30, result=1346269, time=1.420000, gc=0.010000, system=0.0
```

W odpowiedzi system wypisuje argument funkcji, jej wynik, czas działania procesora, czas odśmiecania i czas zużyty przez system operacyjny. Funkcja `time` została zdefiniowana tylko dla argumentu `f` typu `int -> int`. Czytelnik zainspirowany poprzednimi przykładami na pewno łatwo poradzi sobie z zaprogramowaniem jej polimorficznej wersji.

Morał. Kodowanie programu (tworzenie tekstu źródłowego w ustalonym języku programowania) powinno być poprzedzone jego starannym zaprojektowaniem. Projekt programu tworzy się na podstawie analizy problemu, który ma on rozwiązywać. Należy przy tym zadać pytanie, *co jest istotne w sformułowaniu tego problemu*, co zaś jest drugorzędnym detalem, tj. należy dokonać *abstrakcji* problemu, przedstawić go w możliwie ogólnej formie. Abstrakcyjne sformułowanie jest prostsze od oryginalnego, ponieważ nie zawiera drugorzędnych, technicznych detali. Często również rozwiązanie uogólnionego problemu jest łatwiejsze do opracowania. Wykorzystując *polimorfizm* możemy to uogólnione rozwiązanie również zaprogramować w abstrakcyjnej postaci. Rozwiązanie wyjściowego problemu otrzymamy jako oczywiste, czesto jednolinijkowe *ukonkretnienie* rozwiązania ogólnego. Takie podejście znacznie zwiększa czytelność programu i jego odporność na błędy i ułatwia analizę problemu oraz poszukiwanie efektywnego algorytmu. Nadto każdy algorytm programowany jest tylko raz, zgodnie z ideą *code reuse*, co znacznie skraca rozmiar programu.

Zadanie 10.1. Wyznacz typy wszystkich zaprogramowanych w bieżącym rozdziale funkcji.

Rozdział 11

Semantyka języków programowania

W rozdziale 3 omówiliśmy metody ścisłego, matematycznego opisu budowy (składni) programów. Obecnie podamy równie ścisłe metody opisu ich *znaczenia* (*semantyki*). Metody te przedstawimy na przykładzie konkretnego języka programowania. Ponieważ języki używane w praktyce są zbyt wielkie, by służyły za dobry przykład, wykorzystamy w tym celu język D przedstawiony w podrozdziale 3.11. Czytelnik pragnący zapoznać się z formalnymi definicjami „prawdziwych" języków programowania zechce zajrzeć np. do pracy [118], w której podano strukturalną semantykę operacyjną języka SML lub [71], w której opisano semantykę aksjomatyczną języka Pascal.

11.1. Maszyna abstrakcyjna i jej formalny opis

Program pisze się po to, by został *wykonany* na pewnej *maszynie*. Można więc zadać znaczenie programu opisując, co się będzie działo z maszyną wykonującą taki program. Pierwszą decyzję, jaką należy podjąć podczas definiowania semantyki języka, to wybór maszyny, która posłuży do opisania znaczenia programów. Można by spróbować wykorzystać jakiś popularny procesor (Sextium?), jednak po pierwsze architektura takich maszyn nie pasuje zwykle do struktury języka, którego semantykę definiujemy, przez co efekt wykonania programu wyrażony w terminach maszyny nie przekłada się w prosty sposób na opis wykonania tego programu w terminach definiowanego języka. Po drugie taki opis uzależnia nas od jednej konkretnej maszyny. Gdybyśmy chcieli napisać kompilator opisywanego języka na inną maszynę o innej architekturze, taka definicja semantyki byłaby bardzo kłopotliwa. Dlatego lepiej zdefiniować pojęcie *maszyny abstrakcyjnej* danego języka, będącej z jednej strony abstrakcją prawdziwych procesorów, z drugiej zaś dopasowanej swoją strukturą do języka, którego semantykę definiujemy.

Uwagi. Można napisać kompilator K języka programowania J na konkretną maszynę cyfrową M i powiedzieć, że efekt wykonania programu P (jego znaczenie) jest właśnie taki, jak efekt wykonania programu P skompilowanego kompilatorem K i uruchomionego na maszynie M. Taki sposób zadania semantyki języka J nazywa się *definiowaniem przez implementację*. Nieszczęścia jakie wynikają z takiej definicji dobrze ilustruje historia języka LISP (zob. [108, 161]). Koncepcja języka LISP i jego pierwszy interpreter na (lampową) maszynę IBM 704 zostały opracowane w latach 1956–59 w MIT przez grupę pod kierunkiem Johna McCarthy'ego. We wczesnym okresie rozwoju języka LISP nie tworzono samego języka — tworzono raczej jego interpreter, program na konkretną maszynę IBM 704. W języku LISP przetwarza się listy, tj. rekurencyjne struktury danych składające się z *głowy* i *ogona* (ang. *head* i *tail*) lub inaczej elementu pierwszego i listy pozostałych elementów (ang. *first* i *rest*). Pamięć maszyny IBM 704 składała się z co najwyżej 32K słów 36-bitowych. Ze względu na sposób adresowania, którego nie będziemy tutaj szczegółowo omawiać, w słowie maszyny wyróżniono dwie 15-bitowe części, zwane *address* i *decrement*. Maszyna posiadała specjalne rozkazy umożliwiające dostęp do każdej z tych części słowa. Implementując struktury listowe postanowiono w polach *address* i *decrement* przechowywać dwa 15-bitowe adresy głowy i ogona listy. Odpowiednie rozkazy maszynowe pozwalały na łatwy dostęp do obu adresów. Dlatego operację ujawnienia adresu głowy listy nazwano CAR (ang. *Contents of Address Register*), zaś operację ujawnienia adresu ogona listy CDR (ang. *Contents of Decrement Register*). John McCarthy w pierwszej definicji języka [106] napisał:

> The names "car" and "cdr" will come to have mnemonic significance only when we discuss the representation of the system in the computer.

Wkrótce wszyscy z wyjątkiem implementatorów języka zapomnieli, od czego pochodziły te skróty. Steve Russell, jeden z twórców pierwszego interpretera LISP-u opisał tę historię na liście dyskusyjnej `alt.folklore.computers`:

> I wrote the first implemenation of a LISP interpreter on the IBM 704 at MIT in early in 1959. I hand-compiled John McCarthy's "Universal LISP Function".
>
> The 704 family (704, 709, 7090) had "Address" and "Decrement" fields that were 15 bits long in some of the looping instructions. There were also special load and store instructions that moved these 15-bit addresses between memory and the index registers (3 on the 704, 7 on the others). We had devised a representation for list structure that took advantage of these instructions. Because of an *unfortunate temporary lapse of inspiration*, we couldn't think of any other names for the 2 pointers in a list node than "address" and "decrement", so we called the functions CAR for "Contents of Address of Register" and CDR for "Contents of Decrement of Register".
>
> After several months and giving a few classes in LISP, we realized that "first" and "rest" were better names, and we (John McCarthy, I and some of the rest of the AI Project) tried to get people to use them instead. *Alas, it was too late!* We couldn't make it stick at all. So we have CAR and CDR.
>
> As the 704 has 36 bit words, there were 6 bits in the list nodes that were not used. Our initial implementation did not use them at all, but the first garbage collector, commissioned in the summer of 1959, used some of them as flags. Atoms were indicated by having the special value of all 1's in CAR of the first word of the property list. All 0's was NIL, the list terminator. We were attempting to improve on "IPL-V" (for Interpretive Processing of Lists — version 5) which ran on a 650. I believe that the 0 list terminator was used there, but I believe that the all 1's flag for atoms was original.

Nasuwa się pytanie, dlaczego adres głowy listy był przechowywany w lewej części słowa, zaś adres ogona — w prawej. Steve Russell odpowiedział:

> The first implementations of LISP were on the IBM 704, the vacuum-tube ancestor of the vacuum-tube 709, the ancestor of the transistorized 7090, later upgraded to the 7094. The time was early in 1959.
>
> I believe that we started writing list structures in the "natural" (to us English-speakers) from left to right before we had fixed on implementation details on the 704. (In fact, I believe that IPL-V wrote them that way).
>
> I don't remember how we decided to use the address for the first element, but I suspect it had to do with guessing that it would be referenced more, and that there were situations where a memory cycle would be saved when the pointer was in the address.

W maszynie IBM 7094 zamieniono pola *address* i *decrement* miejscami, mimo to implementatorzy LISP-u dalej przechowywali głowę listy w lewej części słowa a ogon w prawej. Teraz jednak adres głowy listy znajdował się w części *decrement*, jednak operacja ujawnienia głowy nadal (ze względu na zgodność z poprzednimi wersjami

języka) nazywała się CAR! Po dzień dzisiejszy we wszystkich popularnych dialektach LISP-u operacje ujawnienia głowy i ogona listy nazywają się CAR i CDR.

Dużo poważniejszy problem był związany z tzw. *wiązaniem parametrów*. Otóż w pierwszym interpreterze LISP-u, inaczej niż w większości języków programowania (wyjątkiem jest TEX) nazwę zmiennej z miejscem jej deklaracji wiązało się *dynamicznie* a nie *statycznie* (zob. rozdział 7). Wszyscy późniejsi implementatorzy LISP-u uznali, że jest to własność celowo wprowadzona do języka. Tymczasem John McCarthy napisał dwadzieścia lat później [108]:

> In all innocence, James R. Slagle programmed the following LISP function definition and complained when it didn't work right:
>
> > The object of the function is to find a subexpression of x satisfying $p[x]$ and return $f[x]$. If the search is unsuccessful, then the continuation function $u[]$ of no arguments is to be computed and its value returned. The difficulty was that when an inner recursion occurred, the value of car$[x]$ wanted was the outer value, but the inner value was actually used. In modern terminology, lexical scoping was wanted, and dynamic scoping was obtained.
>
> *I must confess that I regarded this difficulty as just a bug* and expressed confidence that Steve Russell would soon fix it. He did fix it *but* by inventing the so-called FUNARG device that took the lexical environment along with the functional argument. Similar difficulties later showed up in Algol 60, and Russell's turned out to be one of the more comprehensive solutions to the problem. While it worked well in the interpreter, comprehensiveness and speed seem to be opposed in compiled code, and *this led to a succession of compromises*.

Statyczne wiązanie parametrów wprowadzili dopiero w 1975 roku G. J. Sussman i G. L. Steele, Jr. w dialekcie LISP-u zwanym Scheme [46]. Pojawiło się ono też w popularnym Common LISP-ie.

Jak widać definiowanie języka poprzez napisanie jego implementacji na konkretną maszynę jest bardzo kłopotliwe. Zbyt wiele detali związanych z realizacją kompilatora i błędy w programie kompilatora przenikają do definicji języka. Lepiej więc zdefiniować maszynę abstrakcyjną, niezależną od architektury rzeczywistych procesorów.

Opiszemy teraz maszynę abstrakcyjną języka D. Dla uproszczenia zajmiemy się najpierw jedynie podzbiorem języka D nie zawierającym instrukcji wejścia/wyjścia. Zatem i jego maszyna abstrakcyjna nie będzie musiała zawierać żadnych urządzeń wejścia/wyjścia. Maszyna ta wykonuje obliczenia na *liczbach całkowitych*. Przyjmujemy więc, że *dziedziną interpretacji* wartości całkowitoliczbowych jest zbiór liczb całkowitych $\mathbb{Z}$ (dokonujemy tu idealizacji przyjmując, że maszyna może przetwarzać *dowolnie wielkie* liczby całkowite). Każda implementacja języka narzuca pewne ograniczenia na wielkość tych liczb. Podobnie zbiorem wartości logicznych jest $\mathbb{B} = \{T, F\}$. Liczby całkowite są przechowywane w pamięci. Komórki pamięci mają swoje adresy. Nie jest dla nas istotne, jakiej są one postaci. Przyjmujemy zatem, że jest dany pewien nieskończony zbiór *adresów* $\mathbb{A}$ (dokonujemy tutaj kolejnej idealizacji przyjmując, że pamięć maszyny jest nieograniczona). Zakładamy, że każdej nazwie komórki pamięci X występującej w programie napisanym w języku D jest przyporządkowany pewien adres addr(X) w taki sposób, że różnym nazwom odpowiadają różne adresy.[1] Maszyna abstrakcyjna języka D składa się z pamięci (ang. *store*) i jednostki

[1] Zatem *identyfikatorom* w składni konkretnej odpowiadają *nazwy komórek pamięci* w składni abstrakcyjnej, zaś *nazwom komórek pamięci* w składni abstrakcyjnej odpowiadają *adresy* w kodzie wynikowym programu. Oba powyższe odwzorowania są zwykle różnowartościowe (wyjątkiem jest tzw. *aliasing*, np. dyrektywa EQUIVALENCE w FORTRAN-ie). Pierwsze z odwzorowań zadaje parser podczas analizy składniowej programu w postaci tzw. *tablicy symboli*. Jest to przeważnie tablica z kodowaniem mieszającym (ang. *hashing table*). Nazwy komórek pamięci są zwykle reprezentowane w kompilatorze jako krótkie liczby całkowite bez znaku. Drugie odwzorowanie ustala algorytm przydziału pamięci podczas generowania kodu wynikowego programu. Postać adresów zależy od archi-

sterującej, która czyta abstrakcyjne drzewo rozbioru programu w języku D i wykonuje go zmieniając stan pamięci. Istnieje pewna analogia pomiędzy maszyną abstrakcyjną języka D i automatem skończonym (zob. paragraf 3.7.1). Automat skończony czyta słowo wejściowe i zmienia swój stan pod wpływem przeczytanego słowa. Podobnie maszyna abstrakcyjna języka D czyta abstrakcyjne drzewo rozbioru programu i odpowiednio zmienia swój stan. Maszyna abstrakcyjna nie jest jednak skończona, zawiera bowiem nieskończenie wiele komórek pamięci zawierających dowolnie wielkie liczby, może się zatem znajdować w jednym z nieskończonej liczby stanów. Stan pamięci jest jednoznacznie określony przez podanie zawartości każdej z jej komórek. Może być więc opisany za pomocą funkcji $\sigma : \mathbb{A} \to \mathbb{Z}$, przypisującej nazwom komórek pamięci bieżącą zawartość tych komórek. Przestrzenią stanów pamięci jest zatem zbiór

$$\mathbb{S} = \mathbb{Z}^{\mathbb{A}} = \{\sigma : \mathbb{A} \to \mathbb{Z}\}$$

Ponieważ założyliśmy, że nie rozważamy instrukcji wejścia/wyjścia, stan całej maszyny jest jednoznacznie określony poprzez stan jej pamięci.

W definicjach wielu starszych języków (np. Algolu 60, Algolu 68 i in.) opisywano abstrakcyjną maszynę danego języka, a następnie zadawano znaczenie programu w sposób nieformalny, opisując słownie, jak się zmienia stan takiej maszyny w trakcie wykonywania programu. Język naturalny jest jednak niejednoznaczny. Lepiej użyć formalizmu matematycznego. *Semantyka formalna* to metoda ścisłego, matematycznego określenia, jak zmienia się stan maszyny abstrakcyjnej na skutek wykonania programu. Wyróżnia się następujące rodzaje semantyki formalnej:

Semantyka denotacyjna. Programom przyporządkowuje się pewne obiekty matematyczne, zwane ich *interpretacjami* lub *denotacjami*, stąd nazwa tego rodzaju semantyki. Przeważnie są to funkcje. W przypadku języka D będą to odwzorowania $f : \mathbb{S} \to \mathbb{S}$ określające, jak zmienia się stan maszyny na skutek wykonania programu (tj. takie, że jeśli $\sigma \in \mathbb{S}$ jest stanem maszyny przed wykonaniem programu, którego denotacją jest f, to $f(\sigma)$ jest stanem maszyny po wykonaniu tego programu). W tym celu definiuje się *ad hoc* pewną algebrę nad ustaloną sygnaturą, zwaną *modelem* opisywanego języka programowania, a język traktuje się jako zbiór termów nad tą sygnaturą. Znaczenie programu jest jego interpretacją (jako termu) w modelu. Jest to klasyczna metoda nadawania znaczenia wyrażeniom języka, inspirowana technikami znanymi z *teorii modeli*, tj. działu logiki matematycznej, w którym nadaje się w ten sposób znaczenie formułom logicznym. Metodę denotacyjną opisu języków programowania wynalazł w latach 60-tych zeszłego wieku Christopher Strachey. Na potrzeby semantyki denotacyjnej Dana Scott stworzył tzw. *teorię dziedzin*, tj. matematyczną teorię budowy modeli języków programowania.

Semantyka algebraiczna. Model języka programowania zadaje się za pomocą zbioru aksjomatów równościowych. Jest to technika zbliżona do sposobu, w jaki definiuje się i

tektury maszyny i budowy systemu operacyjnego. Są to przeważnie adresy względne, zorganizowane tak, by kod wynikowy był *przesuwalny*, tj. by mógł być umieszczony w dowolnym miejscu pamięci.

bada różne struktury (np. grupy) w algebrze ogólnej, stąd nazwa tego rodzaju semantyki.

Semantyka operacyjna. To podejście najsilniej akcentuje rolę maszyny abstrakcyjnej języka. Definiuje się pojęcie podobne do tego, które mówiąc o automatach skończonych nazwaliśmy *funkcją przejścia automatu*. Znaczenie programu opisuje się podając, jakie *operacje* wykona maszyna realizująca program, stąd nazwa tego rodzaju semantyki. We współczesnej formie formalizm ten został rozwinięty przez Gordona Plotkina w latach 70-tych zeszłego wieku w postaci tzw. *strukturalnej semantyki operacyjnej*.

Semantyka aksjomatyczna. Definiuje się specjalny formalizm logiczny do wyrażania własności programów i system wnioskowania do dowodzenia tych własności. Język jest wówczas opisany przez zbiór odpowiednich *aksjomatów* i reguł wnioskowania, stąd nazwa tego rodzaju semantyki. Stworzyli ją w latach 60-tych zeszłego wieku R. W. Floyd i C. A. R. Hoare. Semantyka aksjomatyczna bywa też nazywana *logiką Floyda-Hoare'a*.

11.2. Semantyka denotacyjna

W podrozdziale 8.6 opisaliśmy język D jako zbiór termów nad odpowiednio dobraną sygnaturą Σ_D. Zadaliśmy więc składnię języka D definiując sygnaturę Σ_D. Możemy teraz zadać jego semantykę, definiując odpowiednią algebrę $\mathfrak{M} = \langle \{M^a\}_{a\in\mathcal{S}}, \cdot^{\mathfrak{M}} \rangle$ o sygnaturze Σ_D, tj. budując *model* języka D. Każdy term (program) będzie miał wówczas swoją interpretację w algebrze $\mathfrak{M}$.

Ponieważ zbiór $\mathcal{S} = \{A, B, C, I\}$ zawiera cztery rodzaje, musimy wpierw podać cztery dziedziny algebry $\mathfrak{M}$. Wartość wyrażenia arytmetycznego jest liczbą całkowitą zależną od stanu pamięci $\mathbb{S}$. Jego *interpretacja* (*denotacja*) jest więc funkcją przekształcającą stany pamięci w liczby całkowite. Termy gatunku A będziemy zatem interpretować jako funkcje należące do zbioru $M^A = \mathbb{Z}^{\mathbb{S}}$. Podobnie dziedziną interpretacji wyrażeń logicznych będzie $M^B = \mathbb{B}^{\mathbb{S}}$. Instrukcje zmieniają stan maszyny, ich interpretacjami będą więc funkcje przekształcające stany w stany, czyli $M^C = \mathbb{S}^{\mathbb{S}}$. Ostatecznie interpretacjami nazw komórek pamięci będą adresy, mamy więc $M^I = \mathbb{A}$.

Dla każdego symbolu sygnatury musimy teraz podać jego interpretację $\cdot^{\mathfrak{M}}$. Dla przykładu wartość stałej `0` wynosi 0 niezależnie od stanu maszyny. Zatem interpretacją stałej `0` jest funkcja stała, przekształcająca dowolny stan maszyny w liczbę 0, tj. $\mathtt{0}^{\mathfrak{M}}(\sigma) = 0$ dla każdego $\sigma \in \mathbb{S}$. Podobnie interpretacją binarnego symbolu `if` jest funkcja $\mathtt{if}^{\mathfrak{M}} : M^B \times M^C \to M^C$, tj. $\mathtt{if}^{\mathfrak{M}} : \mathbb{B}^{\mathbb{S}} \times \mathbb{S}^{\mathbb{S}} \to \mathbb{S}^{\mathbb{S}}$, taka, że $\mathtt{if}^{\mathfrak{M}}(f, g)(\sigma) = \sigma$, jeśli $f(\sigma) = F$ i $\mathtt{if}^{\mathfrak{M}}(f, g)(\sigma) = g(\sigma)$, jeśli $f(\sigma) = T$.

Dla zadanej funkcji f napis $f[a/n]$ oznacza funkcję f „poprawioną" w punkcie a:

$$f[a/n](b) = \begin{cases} f(b), & \text{gdy } a \neq b \\ n, & \text{w p.p.} \end{cases}$$

Ponadto $f[a_1/n_1, \ldots, a_k/n_k]$ oznacza $f[a_1/n_1] \ldots [a_k/n_k]$. Instrukcja przypisania `X = e` zmienia zawartość komórki pamięci o adresie $a = \mathrm{addr}(X)$ przypisując jej wartość $f(\sigma)$,

gdzie $f = [\![e]\!]^{\mathfrak{M}}$ jest interpretacją wyrażenia e w algebrze $\mathfrak{M}$. Denotacją symbolu przypisania = powinna być zatem funkcja dwu zmiennych (bo symbol = jest binarny) dostarczająca w wyniku funkcję, która dowolnemu stanowi pamięci przyporządkowuje stan pamięci „poprawiony" dla adresu a, tj. taka, że $=^{\mathfrak{M}}(a,\ f) = g$, gdzie $g(\sigma) = \sigma[a/f(\sigma)]$.

Pełna definicja interpretacji symboli sygnatury Σ_D w algebrze $\mathfrak{M}$ jest przedstawiona w tablicach 11.1 na następnej stronie i 11.2 na stronie 210. Na uwagę zasługuje interpretacja symbolu wyłuskania (dereferencji) $i \in \Sigma_D^{I \to A}$. Powiedzieliśmy, że interpretacją nazwy komórki pamięci X jest pewien adres $X^{\mathfrak{M}} = \mathrm{addr}(X)$. Jeżeli X pojawia się w wyrażeniu arytmetycznym, to pragniemy interpretować go jako liczbę całkowitą będącą zawartością komórki pamięci o adresie $\mathrm{addr}(X)$, a nie jako ten adres. Interpretacją symbolu i jest więc funkcja $i^{\mathfrak{M}} : \mathbb{A} \to \mathbb{Z}^{\mathbb{S}}$, taka, że $i^{\mathfrak{M}}(a)(\sigma) = \sigma(a)$.

Dotychczas zakładaliśmy, że interpretacje symboli funkcyjnych w algebrze są funkcjami całkowitymi. Tymczasem interpretacja symbolu / dana wzorem

$$/^{\mathfrak{M}}(f,g)(\sigma) \quad = \quad f(\sigma)\ /\ g(\sigma)$$

nie jest określona w sytuacji, gdy $g(\sigma) = 0$. Aby pozostać w zgodzie z dotychczasową definicją algebry możemy do zbioru liczb całkowitych $\mathbb{Z}$ dodać nowy element $\perp$, reprezentujący *wartość nieokreśloną* i zdefiniować interpretację symbolu / następująco:

$$/^{\mathfrak{M}}(f,g)(\sigma) \quad = \quad \begin{cases} f(\sigma)\ /\ g(\sigma) & \text{gdy } f(\sigma) \neq \perp \text{ i } g(\sigma) \notin \{\perp, 0\} \\ \perp & \text{w p.p.} \end{cases}$$

W podobny warunkowy sposób musimy też wówczas określić interpretację pozostałych operatorów arytmetycznych. Co więcej, element reprezentujący wartość nieokreśloną musimy też dodać do zbiorów wartości logicznych i stanów pamięci i interpretacje wszystkich pozostałych symboli funkcyjnych również rozszerzyć w ten sposób. Aby więc nie komplikować definicji przyjmujemy, że interpretacje symboli funkcyjnych w modelu $\mathfrak{M}$ mogą być funkcjami częściowymi.

Interpretacja symbolu `while`:

$$\texttt{while}^{\mathfrak{M}}(f,g)(\sigma) \quad = \quad \begin{cases} \texttt{while}^{\mathfrak{M}}(f,g)(g(\sigma)), & \text{gdy } f(\sigma) = T \\ \sigma, & \text{w p.p.} \end{cases} \tag{11.1}$$

jest zadana zależnością rekurencyjną. Pojawia się zatem pytanie, czy ta definicja jest poprawna i co ona znaczy. Zwykle nie zastanawiamy się nad poprawnością definicji rekurencyjnych, takich jak np. definicja funkcji silnia:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \end{aligned}$$

ponieważ wydaje się *oczywiste*, że *istnieje dokładnie jedna funkcja* $! : \mathbb{N} \to \mathbb{N}$ *spełniająca oba powyższe równania* (zakładamy, że zbiór liczb naturalnych $\mathbb{N}$ zawiera liczbę zero). Nie zawsze jest to prawdą! Rozważmy rekurencyjną „definicję" funkcji

$$f(n) \quad = \quad f(n-2) + 2 \tag{11.2}$$

$$f(0) \quad = \quad 0 \tag{11.3}$$

$$
\begin{array}{rcl}
\texttt{0}^{\mathfrak{M}}(\sigma) & = & 0 \\
\texttt{1}^{\mathfrak{M}}(\sigma) & = & 1 \\
 & \vdots & \\
X^{\mathfrak{M}} & = & \mathrm{addr}(X) \\
i^{\mathfrak{M}}(a)(\sigma) & = & \sigma(a) \\
\texttt{-}^{\mathfrak{M}}(f)(\sigma) & = & -f(\sigma) \\
\texttt{+}^{\mathfrak{M}}(f,g)(\sigma) & = & f(\sigma)+g(\sigma) \\
\texttt{-}^{\mathfrak{M}}(f,g)(\sigma) & = & f(\sigma)-g(\sigma) \\
\texttt{*}^{\mathfrak{M}}(f,g)(\sigma) & = & f(\sigma)\times g(\sigma) \\
\texttt{/}^{\mathfrak{M}}(f,g)(\sigma) & = & f(\sigma)\ /\ g(\sigma), \quad g(\sigma)\neq 0 \\
\texttt{\%}^{\mathfrak{M}}(f,g)(\sigma) & = & f(\sigma) \bmod g(\sigma), \quad g(\sigma)\neq 0 \\
\texttt{==}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)=g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{!=}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)\neq g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{<}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)<g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{>}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)>g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{<=}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)\leq g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{>=}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)\geq g(\sigma), \\ F & \text{w p.p.} \end{cases} \\
\texttt{true}^{\mathfrak{M}}(\sigma) & = & T \\
\texttt{false}^{\mathfrak{M}}(\sigma) & = & F \\
\texttt{!}^{\mathfrak{M}}(f)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)=F, \\ F, & \text{w p.p.} \end{cases} \\
\texttt{||}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)=T \text{ i } g(\sigma)=T, \\ F, & \text{w p.p.} \end{cases} \\
\texttt{\&\&}^{\mathfrak{M}}(f,g)(\sigma) & = & \begin{cases} T, & \text{gdy } f(\sigma)=T \text{ lub } g(\sigma)=T, \\ F, & \text{w p.p.} \end{cases}
\end{array}
$$

Tablica 11.1. Interpretacja symboli funkcyjnych w modelu $\mathfrak{M}$ języka D (cz. 1)

$$
\begin{aligned}
\texttt{skip}^{\mathfrak{M}}(\sigma) &= \sigma \\
{=}^{\mathfrak{M}}(a, f)(\sigma) &= \sigma[a/f(\sigma)] \\
{;}^{\mathfrak{M}}(f, g)(\sigma) &= f(g(\sigma)) \\
\texttt{if}^{\mathfrak{M}}(f, g)(\sigma) &= \begin{cases} g(\sigma), & \text{gdy } f(\sigma) = T \\ \sigma, & \text{w p.p.} \end{cases} \\
\texttt{if}^{\mathfrak{M}}(f, g_1, g_2)(\sigma) &= \begin{cases} g_1(\sigma), & \text{gdy } f(\sigma) = T \\ g_2(\sigma), & \text{w p.p.} \end{cases} \\
\texttt{while}^{\mathfrak{M}}(f, g)(\sigma) &= \begin{cases} \texttt{while}^{\mathfrak{M}}(f, g)(g(\sigma)), & \text{gdy } f(\sigma) = T \\ \sigma, & \text{w p.p.} \end{cases}
\end{aligned}
$$

Tablica 11.2. Interpretacja symboli funkcyjnych w modelu $\mathfrak{M}$ języka D (cz. 2)

Istnieje *nieskończenie wiele* funkcji $f : \mathbb{N} \to \mathbb{N}$ spełniających powyższe dwa równania, wśród nich np. funkcja identyczności, albo funkcja

$$
f(n) = \begin{cases} n, & \text{dla } n \text{ parzystych} \\ n+1, & \text{dla } n \text{ nieparzystych} \end{cases}
$$

(opisz rodzinę wszystkich funkcji spełniających te dwa równania). Żadna z nich nie jest ani lepsza, ani gorsza! Jeśli dopuścimy funkcje częściowe (określone na właściwym podzbiorze $\mathbb{N}$) i przyjmiemy, że w równaniu (11.2) zmienna n przebiega dziedzinę określoności funkcji f, to do tej dziedziny *muszą* należeć wszystkie liczby parzyste (inne *mogą*, ale nie *muszą*). Wówczas rozwiązaniem układu równości (11.2) i (11.3) będzie też funkcja

$$
f(n) = \begin{cases} n, & \text{dla } n \text{ parzystych} \\ \text{nieokreślona}, & \text{dla } n \text{ nieparzystych} \end{cases}
$$

Ma ona ciekawą własność: zawiera się (jako relacja określona na $\mathbb{N} \times \mathbb{N}$) w każdej innej funkcji spełniającej równania (11.2) i (11.3) (udowodnij to przez indukcję). Nie zamierzamy tutaj rozwijać teorii równań rekurencyjnych. Powiemy jedynie, że dla równania (11.1) istnieje jego najmniejsze rozwiązanie.[2] Przyjmujemy więc, że interpretacje symboli sygnatury Σ_D są najmniejszymi rozwiązaniami równań definiujących interpretacje symboli funkcyjnych. Są to funkcje częściowe, nieokreślone dla niektórych kombinacji argumentów. Zatem także interpretacja termów w modelu $\mathfrak{M}$ nie będzie określona dla wszystkich argumentów, gdyż np. dziedziną określoności funkcji $[\![\texttt{while (true) skip}]\!]^{\mathfrak{M}}$ jest zbiór pusty, zaś dziedzina funkcji $[\![\texttt{2 / } X]\!]^{\mathfrak{M}}$ nie zawiera tych stanów $\sigma \in \mathbb{S}$, dla których $\sigma(\text{addr}(X)) = 0$.

Skoro zadaliśmy algebrę $\mathfrak{M}$, to każde wyrażenie arytmetyczne e, logiczne b i każdy program C (termy stałe gatunków A, B i C nad sygnaturą Σ_D) mają swoją interpretację

[2] Tak nie zawsze być musi. Rozważmy dla przykładu równanie $f(2) = -f(1)$, które ma wiele *nieporównywalnych* rozwiązań minimalnych. Nie ma więc rozwiązania najmniejszego.

(denotację)

$$[\![e]\!]^{\mathfrak{M}} : \mathbb{S} \to \mathbb{Z}, \quad \text{dla } e \in \mathcal{T}^A(\Sigma_D, \emptyset)$$
$$[\![b]\!]^{\mathfrak{M}} : \mathbb{S} \to \mathbb{B}, \quad \text{dla } b \in \mathcal{T}^B(\Sigma_D, \emptyset)$$
$$[\![C]\!]^{\mathfrak{M}} : \mathbb{S} \to \mathbb{S}, \quad \text{dla } C \in \mathcal{T}^C(\Sigma_D, \emptyset)$$

Ponieważ model $\mathfrak{M}$ jest ustalony, jego nazwę będziemy pomijać, pisząc np. $[\![C]\!]$ zamiast $[\![C]\!]^{\mathfrak{M}}$. Interpretację $[\![\cdot]\!]$ termów w modelu $\mathfrak{M}$ będziemy nazywać *funkcją semantyczną*. Zapisując funkcję semantyczną będziemy pomijać nawiasy wokół jej argumentu, pisząc np.

$$[\![X + 2]\!]\sigma \quad \text{zamiast} \quad [\![X + 2]\!](\sigma)$$

W podrozdziale 8.6 powiedzieliśmy, że w praktyce składnię abstrakcyjną języka zadaje się nie przez podanie odpowiedniej sygnatury algebraicznej, lecz za pomocą bezkontekstowej gramatyki abstrakcyjnej (zob. tablica 8.6 na stronie 162). Dlatego również semantykę denotacyjną jest wygodniej zdefiniować podając wprost rekurencyjną definicję funkcji semantycznej. Semantyka denotacyjna wyrażeń języka D w tej postaci jest przedstawiona w tablicy 11.3 na następnej stronie, a jego instrukcji — w tablicy 11.4 na stronie 213.

11.3. Równoważność programów

Mówimy, że programy C_1 i C_2 są *równoważne względem semantyki denotacyjnej*, co oznaczamy $C_1 \sim C_2$, jeżeli $[\![C_1]\!] = [\![C_2]\!]$. Innymi słowy programy C_1 i C_2 są równoważne, jeśli efekt ich wykonania jest zawsze taki sam. Podobnie definiujemy równoważność wyrażeń arytmetycznych i logicznych. Jeżeli udowodnimy, że programy C_1 i C_2 są równoważne, to w dowolnym kontekście jeden z nich możemy zastąpić drugim. Możemy dla przykładu udowodnić przez indukcję względem struktury programu C następujący:

Lemat 11.1. Jeżeli $X \notin \mathrm{Id}(C)$, to $\sigma(X) = [\![C]\!]\sigma(X)$.

w którym $\mathrm{Id}(C)$ oznacza zbiór nazw komórek pamięci występujących w programie C. Lemat ten mówi, że jeśli nazwa komórki pamięci X nie występuje w programie C, to na skutek wykonania tego programu zawartość komórki pamięci o nazwie X nie ulega zmianie. Korzystając z powyższego lematu możemy udowodnić, znów przez indukcję względem programu C:

Twierdzenie 11.2. Jeżeli $\mathrm{Id}(X = e) \cap \mathrm{Id}(C) = \emptyset$, to

$$\texttt{while (}b\texttt{) (}X\texttt{ = }e\texttt{; }C\texttt{)} \quad \sim \quad \texttt{(}X\texttt{ = }e\texttt{; while (}b\texttt{) }C\texttt{)}$$

Mówi ono, że jeżeli program C nie zawiera nazwy X ani żadnej nazwy komórki pamięci występującej w wyrażeniu e, to instrukcję przypisania `X = e` można przesunąć z wnętrza instrukcji `while` przed tę instrukcję bez zmiany znaczenia programu. Taka *optymalizacja pętli* jest często wykonywana przez kompilatory. Dzięki posiadaniu formalnej definicji języka możemy *udowodnić*, że tego rodzaju optymalizacja jest poprawna.

$$
\begin{array}{rcl}
[\![c]\!]\sigma & = & c \\
[\![X]\!]\sigma & = & \sigma(X) \\
[\![e_1 \texttt{+} e_2]\!]\sigma & = & [\![e_1]\!]\sigma + [\![e_2]\!]\sigma \\
[\![e_1 \texttt{-} e_2]\!]\sigma & = & [\![e_1]\!]\sigma - [\![e_2]\!]\sigma \\
[\![e_1 \texttt{*} e_2]\!]\sigma & = & [\![e_1]\!]\sigma \times [\![e_2]\!]\sigma \\
[\![e_1 \texttt{/} e_2]\!]\sigma & = & [\![e_1]\!]\sigma \ / \ [\![e_2]\!]\sigma, \quad [\![e_2]\!]\sigma \neq 0 \\
[\![e_1 \texttt{\%} e_2]\!]\sigma & = & [\![e_1]\!]\sigma \bmod [\![e_2]\!]\sigma, \quad [\![e_2]\!]\sigma \neq 0 \\
[\![e_1 \texttt{==} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma = [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![e_1 \texttt{!=} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma \neq [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![e_1 \texttt{<} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma < [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![e_1 \texttt{>} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma > [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![e_1 \texttt{<=} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma \leq [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![e_1 \texttt{>=} e_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![e_1]\!]\sigma \geq [\![e_2]\!]\sigma \\ F, & \text{w p.p.} \end{cases} \\
[\![\texttt{true}]\!]\sigma & = & T \\
[\![\texttt{false}]\!]\sigma & = & F \\
[\![\texttt{!}b]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![b]\!]\sigma = F \\ F, & \text{w p.p.} \end{cases} \\
[\![b_1 \texttt{||} b_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![b_1]\!]\sigma = T \text{ lub } [\![b_2]\!]\sigma = T \\ F, & \text{w p.p.} \end{cases} \\
[\![b_1 \texttt{\&\&} b_2]\!]\sigma & = & \begin{cases} T, & \text{gdy } [\![b_1]\!]\sigma = T \text{ i } [\![b_2]\!]\sigma = T \\ F, & \text{w p.p.} \end{cases}
\end{array}
$$

Tablica 11.3. Semantyka denotacyjna wyrażeń arytmetycznych i logicznych języka D

$$
\begin{aligned}
[\![\texttt{skip}]\!]\sigma &= \sigma \\
[\![X = e]\!]\sigma &= \sigma[X/[\![e]\!]\sigma] \\
[\![C_1\,;C_2]\!]\sigma &= [\![C_2]\!]([\![C_1]\!]\sigma) \\
[\![\texttt{if}\ (b)\ C]\!]\sigma &= \begin{cases} [\![C]\!]\sigma, & \text{gdy } [\![b]\!]\sigma = T \\ \sigma, & \text{w p.p.} \end{cases} \\
[\![\texttt{if}\ (b)\ C_1\ \texttt{else}\ C_2]\!]\sigma &= \begin{cases} [\![C_1]\!]\sigma, & \text{gdy } [\![b]\!]\sigma = T \\ [\![C_2]\!]\sigma, & \text{w p.p.} \end{cases} \\
[\![\texttt{while}\ (b)\ C]\!]\sigma &= \begin{cases} [\![\texttt{while}\ (b)\ C]\!]([\![C]\!]\sigma), & \text{gdy } [\![b]\!]\sigma = T \\ \sigma, & \text{w p.p.} \end{cases}
\end{aligned}
$$

Tablica 11.4. Semantyka denotacyjna instrukcji języka D

11.4. Semantyka operacyjna

Zamiast wprost definiować funkcje semantyczne, tj. odwzorowania przyporządkowujące elementom pewnych kategorii syntaktycznych (wyrażeniom arytmetycznym, wyrażeniom logicznym, instrukcjom) pewne obiekty matematyczne (ich *denotacje*) możemy opisać zbiór reguł, według których pracuje maszyna abstrakcyjna. Dzięki temu powiemy, jak symulować działanie maszyny abstrakcyjnej. Opiszemy, jakie *operacje* ona wykonuje. Taki sposób zadania semantyki języka programowania nazywa się *semantyką operacyjną*.

Znaczenie konstrukcji języka D zdefiniujemy w sposób modularny — znaczenie złożonej konstrukcji będzie funkcją znaczeń jej składowych. Mówi się, że taka definicja semantyki jest *sterowana składnią*. Jest to łatwe dla tzw. *instrukcji strukturalnych*, tj. takich, w których kolejność obliczeń jest zadana poprzez strukturę programu. Dlatego będziemy mówić o *strukturalnej* semantyce operacyjnej języka D.

Wykonanie instrukcji zmienia stan maszyny. Będziemy zatem zapisywać formuły postaci:

$$\langle C, \sigma_1 \rangle \to \sigma_2$$

Napis $\langle C, \sigma_1 \rangle$ będziemy nazywać *konfiguracją* (maszyny abstrakcyjnej języka D), stan σ_1 będzie *stanem początkowym*, zaś σ_2 *stanem końcowym*. Powyższa formuła znaczy więc „jeżeli maszyna znajduje się w stanie σ_1 i wykona instrukcję C (jeżeli maszyna znajduje się w konfiguracji $\langle C, \sigma_1 \rangle$), to po wykonaniu obliczeń znajdzie się w stanie σ_2”. Opisać semantykę języka, to podać zbiór wszystkich prawdziwych formuł powyższej postaci. Aby móc rozstrzygać, które formuły postaci $\langle C, \sigma_1 \rangle \to \sigma_2$ są prawdziwe, zbudujemy formalny system dowodzenia (zbiór reguł wnioskowania) i powiemy, że prawdziwe są te formuły, które są dowodliwe w naszym systemie. Aby móc określić znaczenie instrukcji, musimy wpierw mieć możliwość określenia znaczenia wyrażeń arytmetycznych i logicznych. Obliczenie wyrażenia nie może zmienić stanu maszyny (nie wpływa na zawartość komórek pamięci). Wynikiem obliczenia wyrażenia arytmetycznego (logicznego) jest natomiast liczba

całkowita (wartość logiczna). Nadto wartość wyrażenia arytmetycznego (logicznego) zależy od stanu maszyny, w którym jest ono obliczane. Dla wyrażeń arytmetycznych e (logicznych b) będziemy zatem pisać formuły postaci $\langle e, \sigma \rangle \to n$ (oraz $\langle b, \sigma \rangle \to p$), gdzie $n \in \mathbb{Z}$ (zaś $p \in \mathbb{B}$). Zbiór reguł wnioskowania będzie *sterowany składnią* w tym sensie, że każdej produkcji gramatyki mówiącej, że dana konstrukcja w języku składa się z pewnych części będzie odpowiadać reguła mówiąca jak zbudować znaczenie tej konstrukcji używając znaczeń jej składowych. Dla przykładu wartością wyrażenia $e_1 + e_2$ jest $n_1 + n_2$, jeśli wartością wyrażenia e_1 jest n_1, zaś wartością wyrażenia e_2 jest n_2:

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2}{\langle e_1 + e_2, \sigma \rangle \to n_1 + n_2}$$

Nie jest to jedna reguła, tylko schemat, za pomocą którego można zbudować nieskończony zbiór reguł, po jednej dla każdej pary liczb całkowitych n_1 i n_2, wyrażeń e_1 i e_2 i pamięci σ. Zestaw reguł wnioskowania dla wyrażeń arytmetycznych jest przedstawiony w tablicy 11.5 na sąsiedniej stronie, dla wyrażeń logicznych w tablicy 11.6 na stronie 216, zaś dla instrukcji w tablicy 11.7 na stronie 217. System wnioskowania określa zatem trzy trójargumentowe relacje

$$\begin{aligned} \langle \cdot, \cdot \rangle \to \cdot \quad &\subseteq \quad \mathcal{A} \times \mathbb{S} \times \mathbb{Z} \\ \langle \cdot, \cdot \rangle \to \cdot \quad &\subseteq \quad \mathcal{B} \times \mathbb{S} \times \mathbb{B} \\ \langle \cdot, \cdot \rangle \to \cdot \quad &\subseteq \quad \mathcal{C} \times \mathbb{S} \times \mathbb{S} \end{aligned}$$

Są to najmniejsze relacje zamknięte względem wszystkich reguł wnioskowania.

Zauważmy, że nie dla wszystkich konfiguracji maszyny $\langle C, \sigma \rangle$ (konfiguracji arytmetycznych $\langle e, \sigma \rangle$, konfiguracji logicznych $\langle b, \sigma \rangle$) istnieje stan σ_1 (liczba całkowita n, wartość logiczna p), takie, że $\langle C, \sigma \rangle \to \sigma_1$ (oraz $\langle e, \sigma \rangle \to n$ i $\langle b, \sigma \rangle \to p$). Np. znaczenie wyrażenia $1 / 0$ jest nieokreślone (dla żadnego stanu maszyny nie ma liczby całkowitej, która byłaby wartością tego wyrażenia w tym stanie), podobnie nieokreślone jest znaczenie programu

```
while (true) skip;
```

(dla żadnego stanu początkowego σ_1 nie istnieje stan końcowy σ_2, taki, że maszyna dojdzie do stanu σ_2 z konfiguracji $\langle$`while (true) skip`$, \sigma_1\rangle$. I słusznie, powyższy program się bowiem zapętla, nie istnieje zatem stan maszyny *po* wykonaniu tego programu.[3]

Przykład 11.3. Pokażemy, że zgodnie z podaną semantyką języka D program

$$\begin{aligned} Z &= X; \\ X &= Y; \\ Y &= Z; \end{aligned}$$

[3] W odróżnieniu jednak od semantyki denotacyjnej poprawność reguły wnioskowania dla pętli `while` nie budzi wątpliwości. Nie definiujemy bowiem funkcji za pomocą zależności rekurencyjnej, tylko relację, mówiąc, że jest to najmniejsza relacja zamknięta względem wszystkich reguł wnioskowania. Najmniejsza relacja zawsze istnieje, jest nią bowiem przekrój wszystkich relacji zamkniętych względem reguł wnioskowania. Nie mamy za to teraz pewności, czy semantyka jest deterministyczna, tj. czy nie istnieje program P i stany σ_1 oraz $\sigma_2 \neq \sigma_3$, takie, że zarówno $\langle P, \sigma_1 \rangle \to \sigma_2$, jak i $\langle P, \sigma_1 \rangle \to \sigma_3$. Poprawność definicji semantyki w tym sensie wymaga osobnego dowodu.

$$\frac{}{\langle c, \sigma \rangle \to c} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2}{\langle e_1 - e_2, \sigma \rangle \to n_1 - n_2}$$

$$\frac{}{\langle X, \sigma \rangle \to \sigma(X)} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2}{\langle e_1 * e_2, \sigma \rangle \to n_1 \times n_2}$$

$$\frac{\langle e, \sigma \rangle \to n}{\langle -e, \sigma \rangle \to -n} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_2 \neq 0}{\langle e_1 \,/\, e_2, \sigma \rangle \to n_1 \,/\, n_2}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2}{\langle e_1 + e_2, \sigma \rangle \to n_1 + n_2} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_2 \neq 0}{\langle e_1 \,\%\, e_2, \sigma \rangle \to n_1 \bmod n_2}$$

Tablica 11.5. Reguły strukturalnej semantyki operacyjnej języka D dla wyrażeń arytmetycznych

powoduje zamianę miejscami zawartości zmiennych X i Y, tj. że

$$\langle Z = X;\ X = Y;\ Y = Z, \sigma \rangle \quad \to \quad \sigma[X/\sigma(Y), Y/\sigma(X), Z/\sigma(X)]$$

dla dowolnego stanu σ. Odpowiedni dowód jest przedstawiony w tablicy 11.8 na stronie 218.

Przykład 11.4. Oto stara sztuczka, pozwalająca zamienić miejscami zawartość zmiennych całkowitych X i Y bez korzystania ze zmiennej pomocniczej:

```
X = X + Y;
Y = X - Y;
X = X - Y;
```

Pokażemy, zgodnie z przyjętą semantyką języka D, że tak jest w istocie, tj. że

$$\langle X = X + Y;\ Y = X - Y;\ X = X - Y, \sigma \rangle \quad \to \quad \sigma[X/\sigma(Y), Y/\sigma(X)]$$

dla dowolnego stanu σ. Odpowiedni dowód jest przedstawiony w tablicy 11.8 na stronie 218.

11.4.1. Obliczanie wyrażeń logicznych

Reguły semantyki operacyjnej języka D dla operatorów `||` i `&&` mówią, że „aby wyznaczyć wartość wyrażenia logicznego $b_1 \otimes b_2$, należy obliczyć wartości logiczne wyrażeń b_1 i b_2, a następnie wyznaczyć wynik, badając która z czterech możliwych kombinacji wartości logicznych zaszła.” Semantyka nie precyzuje, w jakiej kolejności mają być obliczone wyrażenia (dopóki obliczanie wartości wyrażeń nie powoduje skutków ubocznych, nie ma to jednak wpływu na wynik). Taka semantyka wyrażeń logicznych jest przyjęta np. w standardzie języka Pascal. W języku C natomiast, podobnie jak i w SML-u przyjmuje się, że

$$\frac{}{\langle \texttt{true}, \sigma \rangle \to T} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 > n_2}{\langle e_1 \texttt{>} e_2, \sigma \rangle \to T}$$

$$\frac{}{\langle \texttt{false}, \sigma \rangle \to F} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \leq n_2}{\langle e_1 \texttt{>} e_2, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n \quad \langle e_2, \sigma \rangle \to n}{\langle e_1 \texttt{==} e_2, \sigma \rangle \to T} \qquad \frac{\langle b, \sigma \rangle \to T}{\langle \texttt{!}\, b, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \neq n_2}{\langle e_1 \texttt{==} e_2, \sigma \rangle \to F} \qquad \frac{\langle b, \sigma \rangle \to F}{\langle \texttt{!}\, b, \sigma \rangle \to T}$$

$$\frac{\langle e_1, \sigma \rangle \to n \quad \langle e_2, \sigma \rangle \to n}{\langle e_1 \texttt{!=} e_2, \sigma \rangle \to F} \qquad \frac{\langle b_1, \sigma \rangle \to F \quad \langle b_2, \sigma \rangle \to F}{\langle b_1 \texttt{||} b_2, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \neq n_2}{\langle e_1 \texttt{!=} e_2, \sigma \rangle \to T} \qquad \frac{\langle b_1, \sigma \rangle \to F \quad \langle b_2, \sigma \rangle \to T}{\langle b_1 \texttt{||} b_2, \sigma \rangle \to T}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \leq n_2}{\langle e_1 \texttt{<=} e_2, \sigma \rangle \to T} \qquad \frac{\langle b_1, \sigma \rangle \to T \quad \langle b_2, \sigma \rangle \to F}{\langle b_1 \texttt{||} b_2, \sigma \rangle \to T}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 > n_2}{\langle e_1 \texttt{<=} e_2, \sigma \rangle \to F} \qquad \frac{\langle b_1, \sigma \rangle \to T \quad \langle b_2, \sigma \rangle \to T}{\langle b_1 \texttt{||} b_2, \sigma \rangle \to T}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \geq n_2}{\langle e_1 \texttt{>=} e_2, \sigma \rangle \to T} \qquad \frac{\langle b_1, \sigma \rangle \to F \quad \langle b_2, \sigma \rangle \to F}{\langle b_1 \texttt{\&\&} b_2, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 < n_2}{\langle e_1 \texttt{>=} e_2, \sigma \rangle \to F} \qquad \frac{\langle b_1, \sigma \rangle \to F \quad \langle b_2, \sigma \rangle \to T}{\langle b_1 \texttt{\&\&} b_2, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 < n_2}{\langle e_1 \texttt{<} e_2, \sigma \rangle \to T} \qquad \frac{\langle b_1, \sigma \rangle \to T \quad \langle b_2, \sigma \rangle \to F}{\langle b_1 \texttt{\&\&} b_2, \sigma \rangle \to F}$$

$$\frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n_1 \geq n_2}{\langle e_1 \texttt{<} e_2, \sigma \rangle \to F} \qquad \frac{\langle b_1, \sigma \rangle \to T \quad \langle b_2, \sigma \rangle \to T}{\langle b_1 \texttt{\&\&} b_2, \sigma \rangle \to T}$$

Tablica 11.6. Reguły strukturalnej semantyki operacyjnej języka D dla wyrażeń logicznych

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \to \sigma} \qquad \frac{\langle b, \sigma \rangle \to T \quad \langle C, \sigma \rangle \to \sigma'}{\langle \texttt{if (}b\texttt{) } C, \sigma \rangle \to \sigma'}$$

$$\frac{\langle e, \sigma \rangle \to n}{\langle x, \texttt{=}e, \sigma \rangle \to \sigma[x/n]} \qquad \frac{\langle b, \sigma \rangle \to T \quad \langle C_1, \sigma \rangle \to \sigma'}{\langle \texttt{if (}b\texttt{) } C_1 \texttt{ else } C_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle C_1, \sigma \rangle \to \sigma' \quad \langle C_2, \sigma' \rangle \to \sigma''}{\langle C_1\, C_2, \sigma \rangle \to \sigma''} \qquad \frac{\langle b, \sigma \rangle \to F \quad \langle C_2, \sigma \rangle \to \sigma'}{\langle \texttt{if (}b\texttt{) } C_1 \texttt{ else } C_2, \sigma \rangle \to \sigma'}$$

$$\frac{\langle b, \sigma \rangle \to F}{\langle \texttt{if (}b\texttt{) } C, \sigma \rangle \to \sigma} \qquad \frac{\langle b, \sigma \rangle \to F}{\langle \texttt{while (}b\texttt{) } C, \sigma \rangle \to \sigma}$$

$$\frac{\langle b, \sigma \rangle \to T \quad \langle C, \sigma \rangle \to \sigma' \quad \langle \texttt{while (}b\texttt{) } C, \sigma' \rangle \to \sigma''}{\langle \texttt{while (}b\texttt{) } C, \sigma \rangle \to \sigma''}$$

Tablica 11.7. Reguły strukturalnej semantyki operacyjnej języka D dla instrukcji

w wyrażeniu logicznym b_1 `||` b_2 najpierw oblicza się wartość wyrażenia b_1 i, jeśli okaże się ono prawdziwe, nie oblicza się wyrażenia b_2 (bo wynik obliczeń jest już przesądzony). Podobnie nie oblicza się wyrażenia b_2, jeśli w wyrażeniu b_1 `&&` b_2 wyrażenie b_1 jest fałszywe. Taka semantyka operatorów logicznych bywa nazywana *short circuit evaluation* lub (w kręgu języków funkcjonalnych) *wartościowaniem leniwym* (w odróżnieniu od *gorliwego*, w którym najpierw oblicza się oba podwyrażenia, a następnie bada kombinację ich wartości, tak jak w naszej oryginalnej semantyce języka D). Zestaw reguł dla leniwego wartościowania operatorów `||` i `&&` przedstawia tablica 11.9 na stronie 219. W SML-u aby podkreślić fakt, że wyrażenia logiczne są wartościowane leniwie, używa się słów kluczowych `andalso` i `orelse`, zamiast `and` i `or`. W Adzie są oba rodzaje operatorów logicznych: gorliwe `and` i `or` oraz leniwe `and then` i `or else`.

11.5. Równoważność semantyk

Nasuwa się pytanie, czy definiując semantykę operacyjną i denotacyjną języka D opisaliśmy ten sam język. Tak jest w istocie, o czym można się przekonać dowodząc (przez indukcję względem struktury odpowiednich wyrażeń i programów) następujące lematy i twierdzenie:

Lemat 11.5. Dla dowolnego wyrażenia arytmetycznego $e \in \mathcal{A}$, dowolnego stanu σ i dowolnej liczby n zachodzi

$$\langle e, \sigma \rangle \to n \qquad \text{wtw} \qquad [\![e]\!]\sigma = n$$

$$\dfrac{\dfrac{\dfrac{}{\langle X,\sigma\rangle \to m}}{\langle Z = X,\sigma\rangle \to \sigma[Z/m]} \quad \dfrac{\dfrac{}{\langle Y,\sigma[Z/m]\rangle \to n}}{\langle X = Y,\sigma[Z/m]\rangle \to \sigma[Z/m, X/n]}}{\langle Z = X;\ X = Y,\sigma\rangle \to \sigma[X/n, Z/m]} \quad \dfrac{\dfrac{}{\langle Z,\sigma[X/n, Z/m]\rangle \to m}}{\langle Y = Z,\sigma[X/n, Z/m]\rangle \to \sigma[X/n, Y/m, Z/m]}}{\langle Z = X;\ X = Y;\ Y = Z,\sigma\rangle \to \sigma[X/n, Y/m, Z/m]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle X,\sigma\rangle \to m} \quad \dfrac{}{\langle Y,\sigma\rangle \to n}}{\langle X + Y,\sigma\rangle \to m + n}}{\langle X = X + Y,\sigma\rangle \to \sigma[X/m + n]} \quad \dfrac{\dfrac{\dfrac{}{\langle X,\sigma[X/m + n]\rangle \to m + n} \quad \dfrac{}{\langle Y,\sigma[X/m + n]\rangle \to n}}{\langle X - Y,\sigma[X/m + n]\rangle \to m}}{\langle Y = X - Y,\sigma[X/m + n]\rangle \to \sigma[X/m + n, Y/m]}}{\langle X = X + Y;\ Y = X - Y,\sigma\rangle \to \sigma[X/m + n, Y/m]} \quad A}{\langle X = X + Y;\ Y = X - Y;\ X = X - Y,\sigma\rangle \to \sigma[X/n, Y/m]}$$

$$\text{gdzie } A \quad = \quad \dfrac{\dfrac{}{\langle X,\sigma[X/m + n, Y/m]\rangle \to m + n} \quad \dfrac{}{\langle Y,\sigma[X/m + n, Y/m]\rangle \to m}}{\langle X = X - Y,\sigma[X/m + n, Y/m]\rangle \to \sigma[X/n, Y/m]}$$

Tablica 11.8. Przykłady wnioskowań w semantyce operacyjnej (dla skrócenia zapisu $m = \sigma(X)$ i $n = \sigma(Y)$)

$$\frac{\langle b_1, \sigma\rangle \to F \quad \langle b_2, \sigma\rangle \to F}{\langle b_1 \,||\, b_2, \sigma\rangle \to F} \qquad \frac{\langle b_1, \sigma\rangle \to F}{\langle b_1 \,\&\&\, b_2, \sigma\rangle \to F}$$

$$\frac{\langle b_1, \sigma\rangle \to F \quad \langle b_2, \sigma\rangle \to T}{\langle b_1 \,||\, b_2, \sigma\rangle \to T} \qquad \frac{\langle b_1, \sigma\rangle \to T \quad \langle b_2, \sigma\rangle \to F}{\langle b_1 \,\&\&\, b_2, \sigma\rangle \to F}$$

$$\frac{\langle b_1, \sigma\rangle \to T}{\langle b_1 \,||\, b_2, \sigma\rangle \to T} \qquad \frac{\langle b_1, \sigma\rangle \to T \quad \langle b_2, \sigma\rangle \to T}{\langle b_1 \,\&\&\, b_2, \sigma\rangle \to T}$$

Tablica 11.9. Reguły strukturalnej semantyki operacyjnej języka D dla wyrażeń logicznych uwzględniające metodę skróconą obliczania koniunkcji i alternatywy

Lemat 11.6. Dla dowolnego wyrażenia logicznego $b \in \mathcal{B}$, dowolnego stanu σ i dowolnej wartości logicznej $p \in \mathbb{B}$ zachodzi

$$\langle b, \sigma\rangle \to p \qquad \text{wtw} \qquad [\![b]\!]\sigma = p$$

Twierdzenie 11.7. Dla dowolnego programu w $C \in \mathcal{C}$ języku D i dowolnych stanów σ i σ' zachodzi

$$\langle C, \sigma\rangle \to \sigma' \qquad \text{wtw} \qquad [\![C]\!]\sigma = \sigma'$$

11.6. Przykład: semantyka wyrażeń regularnych

Semantykę formalną można zadawać nie tylko dla języków programowania, lecz także dowolnych innych języków formalnych. Dla przykładu rozważmy wyrażenia, których składnia abstrakcyjna jest przedstawiona w tablicy 11.10 na stronie 221. Będziemy je nazywać *wyrażeniami regularnymi z operatorem punktu stałego* nad alfabetem {a, b}. W porównaniu z wyrażeniami regularnymi opisanymi w paragrafie 3.7.4 nie zawierają one operatora domknięcia Kleene'go, mają natomiast zmienne i kwantyfikator μ. Znaczenie tych wyrażeń można zdefiniować w sposób nieformalny następująco:

- $\emptyset$ oznacza język pusty;
- ϵ oznacza język $\{\epsilon\}$ zawierający jedynie pusty napis;
- a oznacza język {a} zawierający jedynie jednoliterowy napis złożony z litery a;
- b oznacza język {b} zawierający jedynie jednoliterowy napis złożony z litery b;
- X oznacza dowolny język (zależnie od interpretacji zmiennych wolnych);
- $e_1 \cdot e_2$ oznacza język napisów postaci uw, gdzie u należy do języka opisanego wyrażeniem e_1, zaś w należy do języka opisanego wyrażeniem e_2;
- $e_1 + e_2$ oznacza sumę mnogościową języków opisanych wyrażeniami e_1 i e_2;

$\mu X.e$ oznacza najmniejszy język L, który zawiera język opisany wyrażeniem e, przy czym interpretacją zmiennej X w wyrażeniu e jest język L (definicja jest rekurencyjna), np. $\mu X.(\mathtt{a} + \mathtt{a} \cdot X)$ reprezentuje nieskończony język $\{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$.

Języki, które można opisać powyższymi wyrażeniami, to języki regularne. Istotnie, każdy język regularny można opisać takim wyrażeniem. Wystarczy w tym celu pokazać, jak „symulować" operator domknięcia Kleene'go i powołać się na twierdzenie 3.6. Zauważmy, że poszukiwanym odpowiednikiem wyrażenia w^* jest $\mu X.(\epsilon + w \cdot X)$. Dowód tego faktu, oraz dowód, że każdy język opisany wyrażeniem z operatorem punktu stałego jest regularny pozostawiamy Czytelnikowi jako ćwiczenie.

Zdefiniujemy formalnie semantykę denotacyjną tego języka wyrażeń. Udowodnimy przy tym, że definicja znaczenia $\mu X.e$ jest poprawna, tj. że zawsze taki najmniejszy zbiór istnieje. Następnie zdefiniujemy formalnie semantykę operacyjną wyrażeń nie zawierających zmiennych wolnych, tj. za pomocą odpowiednich aksjomatów i reguł wnioskowania określimy relację $\cdot \in \cdot$, taką że $w \in e$, gdzie w jest słowem nad alfabetem $\{\mathtt{a}, \mathtt{b}\}$ a e wyrażeniem regularnym, jeśli w należy do języka opisanego wyrażeniem e. Na koniec udowodnimy równoważność obu zdefiniowanych semantyk.

Wpierw zadamy semantykę denotacyjną wyrażeń regularnych. Niech $\eta : \mathbb{V} \to \{\mathtt{a}, \mathtt{b}\}^*$, gdzie $\mathbb{V}$ jest zbiorem zmiennych, oznacza interpretację zmiennych wolnych w wyrażeniach (dowolne odwzorowanie, które zmiennym przyporządkowuje języki). Interpretację wyrażeń definiujemy indukcyjnie względem ich struktury (zob. tablica 11.11 na następnej stronie). Aby pokazać, że semantyka denotacyjna zgadza się z podaną wyżej nieformalną definicją znaczenia wyrażeń regularnych dowodzimy najpierw indukcyjnie, że dla dowolnego wyrażenia e, dowolnej interpretacji zmiennych wolnych η i dowolnej niepustej rodziny języków $\{L_t\}_t$ zachodzi

$$\bigcap_t [\![e]\!]_{\eta[X/L_t]} \quad = \quad [\![e]\!]_{\eta[X/\bigcap_t L_t]} \tag{11.4}$$

Dowód jest standardowy, więc pozostawimy go jako ćwiczenie. Rodzina zbiorów L, takich że

$$L \supseteq [\![e]\!]_{\eta[X/L]} \tag{11.5}$$

jest niepusta, ponieważ np. $\{\mathtt{a}, \mathtt{b}\}^*$ do niej należy. Nadto przekrój dowolnej rodziny zbiorów $\{L_t\}_t$ o własności (11.5) również ją posiada: przyjmijmy, że każdy ze zbiorów L_t pewnej rodziny $\{L_t\}_t$ spełnia własność (11.5), zatem $w \in [\![e]\!]_{\eta[X/L_t]}$ pociąga $w \in L_t$ dla każdego t. Jeśli $w \in \bigcap_t [\![e]\!]_{\eta[X/\bigcap_t L_t]}$, tj. $w \in [\![e]\!]_{\eta[X/\bigcap_t L_t]} = \bigcap_t [\![e]\!]_{\eta[X/L_t]}$ dla każdego t, to $w \in [\![e]\!]_{\eta[X/L_t]}$ dla każdego t, więc $w \in L_t$, dla każdego t i $w \in \bigcap_t L_t$. Zatem $\bigcap_t L_t \supseteq [\![e]\!]_{\eta[X/\bigcap_t L_t]}$. Ponadto przekrój wszystkich zbiorów o własności (11.5) jest oczywiście najmniejszym zbiorem o tej własności. Język $[\![\mu X.e]\!]_\eta$ jest zatem najmniejszym językiem L, takim, że $L \supseteq [\![e]\!]_{\eta[X/L]}$.

Określiliśmy funkcję $[\![\cdot]\!]_\eta$ przyporządkowującą wyrażeniom regularnym ich denotacje w zbiorze wszystkich języków nad alfabetem $\{\mathtt{a}, \mathtt{b}\}$, zadaliśmy więc semantykę denotacyjną.

Dla wyrażeń nie zawierających zmiennych wolnych pokazujemy przez indukcję, że

$$[\![e]\!]_\eta \quad = \quad [\![e]\!]_{\eta'}$$

$$e \quad ::= \quad \emptyset \mid \epsilon \mid \mathtt{a} \mid \mathtt{b} \mid X \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid \mu X.e$$

Tablica 11.10. Składnia abstrakcyjna wyrażeń regularnych

$$\begin{aligned}
[\![\emptyset]\!]_\eta &= \emptyset \\
[\![\epsilon]\!]_\eta &= \{\epsilon\} \\
[\![\mathtt{a}]\!]_\eta &= \{\mathtt{a}\} \\
[\![\mathtt{b}]\!]_\eta &= \{\mathtt{b}\} \\
[\![X]\!]_\eta &= \eta(X) \\
[\![e_1 \cdot e_2]\!]_\eta &= \{uw \mid u \in [\![e_1]\!]_\eta \wedge w \in [\![e_2]\!]_\eta\} \\
[\![e_1 + e_2]\!]_\eta &= [\![e_1]\!]_\eta \cup [\![e_2]\!]_\eta \\
[\![\mu X.e]\!]_\eta &= \bigcap \{L \mid L \supseteq [\![e]\!]_{\eta[X/L]}\}
\end{aligned}$$

Tablica 11.11. Semantyka denotacyjna wyrażeń regularnych

$$\frac{}{\epsilon \in \epsilon} \qquad \frac{u \in e_1 \quad w \in e_2}{uw \in e_1 \cdot e_2}$$

$$\frac{}{\mathtt{a} \in \mathtt{a}} \qquad \frac{u \in e_1}{u \in e_1 + e_2} \qquad \frac{u \in e[X/\mu X.e]}{u \in \mu X.e}$$

$$\frac{}{\mathtt{b} \in \mathtt{b}} \qquad \frac{u \in e_2}{u \in e_1 + e_2}$$

Tablica 11.12. Semantyka operacyjna wyrażeń regularnych

$$\begin{aligned}
X \cdot (Y \cdot Z) &= (X \cdot Y) \cdot Z & \qquad X \cdot \epsilon &= X \\
X + (Y + Z) &= (X + Y) + Z & \qquad \epsilon \cdot X &= X \\
X + Y &= Y + X & \qquad X \cdot \emptyset &= \emptyset \\
(X + Y) \cdot Z &= X \cdot Z + Y \cdot Z & \qquad \emptyset \cdot X &= \emptyset \\
X \cdot (Y + Z) &= X \cdot Y + X \cdot Z & \qquad \mu X.e &= e[X/\mu X.e] \\
X + \emptyset &= X
\end{aligned}$$

Tablica 11.13. Semantyka algebraiczna wyrażeń regularnych

dla dowolnych interpretacji η i η'. Dowód jest standardowy, więc pozostawiamy go jako ćwiczenie. Dla takich wyrażeń interpretację η będziemy pomijać i pisać $[\![e]\!]$.

Semantyka operacyjna jest przedstawiona w tablicy 11.12 na poprzedniej stronie. Jest to zbiór reguł wnioskowania, za pomocą których możemy dowodzić, które słowa należą do języka opisywanego przez wyrażenie regularne. Reguły te możemy traktować jako opis *operacji*, które należy wykonać, by wygenerować każde słowo należące do języka opisywanego przez wyrażenie regularne. Dlatego tę definicję zaliczamy do klasy semantyk operacyjnych.

Na koniec możemy pokazać równoważność obu semantyk, tj. udowodnić przez indukcję względem struktury nie zawierającego zmiennych wolnych wyrażenia e, że

$$w \in [\![e]\!] \quad \Longleftrightarrow \quad w \in e$$

Dowód jest standardowy, więc pozostawimy go jako ćwiczenie.

11.7. Semantyka algebraiczna

Na początku bieżącego rozdziału wspomnieliśmy, że zamiast definiować model języka *ad hoc*, tak jak to ma miejsce w semantyce denotacyjnej, możemy go zadać za pomocą zbioru równości. Choć jest wiele innych możliwości, zwykle przyjmuje się, że modelem języka zadanym przez pewien zbiór równości jest algebra początkowa dla tego zbioru. Jest to tzw. *semantyka algebry początkowej*.

Ze względu na problemy ze zdefiniowaniem w semantyce algebraicznej pojęcia stanu obliczeń, nie zadamy semantyki algebraicznej języka D (Czytelnika zainteresowanego algebraicznym definiowaniem semantyki języków imperatywnych odsyłamy do książki Goguena [62]). W zamian zadamy semantykę algebraiczną wyrażeń regularnych z poprzedniego podrozdziału. Jest ona przedstawiona w tablicy 11.13 na poprzedniej stronie.

Na mocy twierdzenia 8.20 algebrą początkową dla zbioru równości E jest

$$\mathfrak{P} \quad = \quad \langle \mathcal{T}(\Sigma, \emptyset)/_{\sim}, \cdot^{\mathfrak{P}} \rangle$$

gdzie $t \sim s \iff (t = s) \in \mathrm{Th}(E)$ i $f^{\mathfrak{P}}([t_1]_{\sim}, \ldots, [t_n]_{\sim}) = [f(t_1, \ldots, t_n)]_{\sim}$. Pozostawiamy Czytelnikowi jako ćwiczenie dowód faktu, że algebra początkowa dla zbioru równości przedstawionych w tablicy 11.13 jest izomorficzna z algebrą zadaną przez semantykę denotacyjną z tablicy 11.11.

Do semantyki algebraicznej powrócimy w rozdziale 12, w którym mówimy o tzw. *specyfikacjach algebraicznych*.

11.8. Semantyka aksjomatyczna

Do uzupełnienia: Treść tego podrozdziału jest jedynie naszkicowana i wymaga rozszerzenia o dalsze przykłady i komentarze.

11.8.1. Język asercji

Standardowym językiem logiki, służącym do formalizacji matematyki jest język rachunku predykatów pierwszego rzędu ([173], rozdział 12). Możemy wyrażać w nim różne sądy o badanej przez nas rzeczywistości matematycznej. Dla przykładu zdanie $\forall x.x^2 \geq 0$ jest prawdziwe w odniesieniu do zbioru liczb rzeczywistych, fałszywe zaś w zbiorze liczb zespolonych. Przypomnijmy, że termy języka predykatów zbudowane są ze zmiennych i, j itd. stałych n, m itd. i symboli funkcyjnych o ustalonej arności (np. binarnych operatorów $+, -$ itp.). Przy ustalonej interpretacji zmiennych i, j itd. każdemu termowi jest jednoznacznie przyporządkowany pewien element dziedziny interpretacji. Aby móc mówić o zawartości komórek pamięci składnię termów rozszerzamy o nazwy komórek pamięci. Tak rozszerzony język rachunku predykatów pierwszego rzędu nazywamy *językiem asercji*. Abstrakcyjna składnia języka asercji jest przedstawiona w tablicy 11.14.

Dowolną funkcję $I : \mathbb{I} \to \mathbb{Z}$, gdzie $\mathbb{I}$ jest zbiorem zmiennych, nazywamy *interpretacją* zmiennych. Interpretacja zmiennych I wraz ze stanem pamięci σ jednoznacznie zadają interpretację termów $\hat{I}_\sigma : \mathcal{T} \to \mathbb{Z}$ zgodnie z definicją rekurencyjną przedstawioną w tablicy 11.15. Jeżeli asercja ϕ jest prawdziwa przy danej interpretacji zmiennych I w stanie σ, będziemy pisać $I, \sigma \models \phi$. Prawdziwość asercji przy danej interpretacji zmiennych I w stanie σ jest zdefiniowana w tablicy 11.16. Będziemy mówić, że asercja ϕ jest *prawdziwa* i pisać $\models \phi$, jeśli będzie ona prawdziwa przy każdej interpretacji zmiennych i w każdym stanie pamięci.

11.8.2. Reguły częściowej i całkowitej poprawności

Będziemy teraz mówić o programach formuły postaci

$$\{\phi\}C\{\psi\} \quad \text{oraz} \quad [\phi]C[\psi]$$

Formuła $\{\phi\}C\{\psi\}$ mówi, że jeśli w pewnym świecie (w pewnym stanie pamięci, przy pewnej interpretacji zmiennych) jest prawdziwa asercja ϕ, to *jeśli* obliczenie programu C dobiegnie końca, to będzie prawdziwa asercja ψ. Formułę postaci $\{\phi\}C\{\psi\}$ nazywamy *warunkiem częściowej poprawności programu*. Formuła $[\phi]C[\psi]$ mówi, że jeśli w pewnym świecie (w pewnym stanie pamięci, przy pewnej interpretacji zmiennych) jest prawdziwa asercja ϕ, to obliczenie programu C dobiegnie końca i będzie prawdziwa asercja ψ. Formułę postaci $[\phi]C[\psi]$ nazywamy *warunkiem całkowitej poprawności programu*. Reguły częściowej poprawności programów dla języka D zawiera rysunek 11.1, zaś całkowitej tablica 11.17. Tablica 11.19 zawiera przykład dowodu częściowej poprawności programu.

11.8.3. Najsłabsze warunki wstępne

Najsłabsze warunki wstępne (wp, od *weakest preconditions*) Dijkstry są zdefiniowane w tablicy 11.18.

Fakt 11.8. $\psi = \mathrm{wp}(C, \phi)$ wtedy i tylko wtedy, gdy $[\phi]\ C\ [\psi]$ i dla każdego ϕ' takiego, że $[\phi']\ C\ [\psi]$ zachodzi $\phi \Rightarrow \phi'$.

stałe:	n		
zmienne:	i		
nazwy komórek pamięci:	X		
operatory arytmetyczne:	$\otimes$	$=$	$+ \mid - \mid \times \mid / \mid \mathrm{mod}$
operatory relacji:	$\ominus$	$=$	$= \mid \neq \mid < \mid > \mid \leq \mid \geq$
operatory logiczne:	$\oslash$	$=$	$\vee \mid \wedge \mid \Rightarrow \mid \Leftrightarrow$
termy:	$\mathcal{T}$	$=$	$n \mid i \mid X \mid -\mathcal{T} \mid \mathcal{T} \otimes \mathcal{T}$
formuły:	$\mathcal{F}$	$=$	$T \mid F \mid \mathcal{T} \ominus \mathcal{T} \mid \neg\mathcal{F} \mid \mathcal{F} \oslash \mathcal{F} \mid \forall i.\mathcal{F} \mid \exists i.\mathcal{F}$

Tablica 11.14. Język asercji

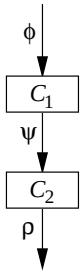
$$
\begin{aligned}
\hat{I}_\sigma(n) &= n \\
\hat{I}_\sigma(i) &= I(i) \\
\hat{I}_\sigma(X) &= \sigma(X) \\
\hat{I}_\sigma(e_1 + e_2) &= \hat{I}_\sigma(e_1) + \hat{I}_\sigma(e_2) \\
\hat{I}_\sigma(e_1 - e_2) &= \hat{I}_\sigma(e_1) - \hat{I}_\sigma(e_2) \\
\hat{I}_\sigma(e_1 * e_2) &= \hat{I}_\sigma(e_1) \times \hat{I}_\sigma(e_2) \\
\hat{I}_\sigma(e_1 \,/\, e_2) &= \hat{I}_\sigma(e_1) \;/\; \hat{I}_\sigma(e_2), \quad \hat{I}_\sigma(e_2) \neq 0 \\
\hat{I}_\sigma(e_1 \,\%\, e_2) &= \hat{I}_\sigma(e_1) \bmod \hat{I}_\sigma(e_2), \quad \hat{I}_\sigma(e_2) \neq 0
\end{aligned}
$$

Tablica 11.15. Interpretacja termów języka asercji

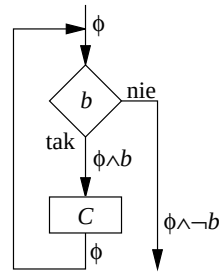
$I, \sigma \models t_1 \ominus t_2$	jeśli $\hat{I}_\sigma(t_1) \ominus \hat{I}_\sigma(t_2)$, gdzie $\ominus$ jest jednym z operatorów $=, \neq, <, >, \leq$ lub $\geq$
$I, \sigma \models \neg\phi$	jeśli nie prawda, że $I, \sigma \models \phi$
$I, \sigma \models \phi \vee \psi$	jeśli $I, \sigma \models \phi$ lub $I, \sigma \models \psi$
$I, \sigma \models \phi \wedge \psi$	jeśli $I, \sigma \models \phi$ i $I, \sigma \models \psi$
$I, \sigma \models \forall i.\phi$	jeśli dla każdej liczby $n \in \mathbb{Z}$ zachodzi $I[i/n], \sigma \models \phi$
$I, \sigma \models \exists i.\phi$	jeśli istnieje liczba $n \in \mathbb{Z}$, taka, że $I[i/n], \sigma \models \phi$

Tablica 11.16. Definicja prawdziwości asercji

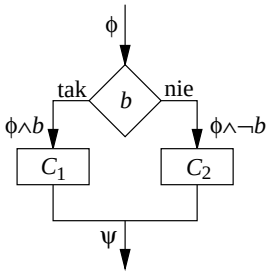
$$\frac{}{\{\phi\}\ \texttt{skip}\ \{\phi\}}\ \text{(skip)} \qquad \frac{}{\{\phi[X/e]\}\ X = e\ \{\phi\}}\ \text{(asgn)}$$



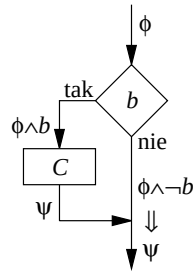
$$\frac{\{\phi\}\ C_1\ \{\psi\} \quad \{\psi\}\ C_2\ \{\rho\}}{\{\phi\}\ C_1 ; C_2\ \{\rho\}}\ \text{(seq)}$$



$$\frac{\{\phi \wedge b\}\ C\ \{\phi\}}{\{\phi\}\ \texttt{while}\ (b)\ C\ \{\phi \wedge \neg b\}}\ \text{(while)}$$



$$\frac{\{\phi \wedge b]\ C_1\ \{\psi\} \quad \{\phi \wedge \neg b\}\ C_2\ \{\psi\}}{\{\phi\}\ \texttt{if}\ (b)\ C_1\ \texttt{else}\ C_2\ \{\psi\}}\ \text{(if-else)}$$



$$\frac{\{\phi \wedge b\}\ C\ \{\psi\} \quad \models \phi \wedge \neg b \Rightarrow \psi}{\{\phi\}\ \texttt{if}\ (b)\ C\ \{\psi\}}\ \text{(if)}$$

$$\frac{\models \phi' \Rightarrow \phi \quad \{\phi\}\ C\ \{\psi\}}{\{\phi'\}\ C\ \{\psi\}}\ \text{(precond)} \qquad \frac{\{\phi\}\ C\ \{\psi\} \quad \models \psi \Rightarrow \psi'}{\{\phi\}\ C\ \{\psi'\}}\ \text{(postcond)}$$

Rysunek 11.1. Reguły częściowej poprawności dla języka D

$$\frac{}{[\phi]\ \texttt{skip}\ [\phi]}\ \text{(skip)}$$

$$\frac{}{[\phi[X/e]]\ X = e\ [\phi]}\ \text{(asgn)}$$

$$\frac{[\phi]\ C_1\ [\psi] \quad [\psi]\ C_2\ [\rho]}{[\phi]\ C_1\,;C_2\ [\rho]}\ \text{(seq)}$$

$$\frac{[\phi \wedge b]\ C\ [\psi] \quad \models \phi \wedge \neg b \Rightarrow \psi}{[\phi]\ \texttt{if}\ (b)\ C\ [\psi]}\ \text{(if)}$$

$$\frac{[\phi \wedge b]\ C_1\ [\psi] \quad [\phi \wedge \neg b]\ C_2\ [\psi]}{[\phi]\ \texttt{if}\ (b)\ C_1\ \texttt{else}\ C_2\ [\psi]}\ \text{(if-else)}$$

$$\frac{[\phi \wedge b \wedge e = i]\ C\ [\phi \wedge e < i] \quad \models \phi \wedge b \Rightarrow e > 0}{[\phi]\ \texttt{while}\ (b)\ C\ [\phi \wedge \neg b]}\ \text{(while)}$$

$$\frac{\models \phi' \Rightarrow \phi \quad [\phi]\ C\ [\psi]}{[\phi']\ C\ [\psi]}\ \text{(precond)}$$

$$\frac{[\phi]\ C\ [\psi] \quad \models \psi \Rightarrow \psi'}{[\phi]\ C\ [\psi']}\ \text{(postcond)}$$

Tablica 11.17. Reguły całkowitej poprawności dla języka D

$$\begin{aligned}
\mathrm{wp}(\texttt{skip}, \phi) &= \phi \\
\mathrm{wp}(x = e, \phi) &= \phi[x/e] \\
\mathrm{wp}(C_1\,;C_2, \phi) &= \mathrm{wp}(C_2, \mathrm{wp}(C_1, \phi)) \\
\mathrm{wp}(\texttt{if}\ (b)\ C, \phi) &= (b \wedge \mathrm{wp}(C, \phi)) \vee (\neg b \wedge \phi) \\
\mathrm{wp}(\texttt{if}\ (b)\ C_1\ \texttt{else}\ C_2, \phi) &= (b \wedge \mathrm{wp}(C_1, \phi)) \vee (\neg b \wedge \mathrm{wp}(C_2, \phi)) \\
\mathrm{wp}(\texttt{while}\ (b)\ C, \phi) &= \bigvee_{i=0}^{\infty} H_i \\
H_0 &= \neg b \wedge \phi \\
H_{i+1} &= b \wedge \mathrm{wp}(C, H_i)
\end{aligned}$$

Tablica 11.18. Najsłabsze warunki wstępne

$$\left\{\mathtt{N} > 0 \wedge \mathtt{X} = k \wedge \mathtt{N} = i\right\}$$
$$\left\{\mathtt{X} = k^{2^{i-\mathtt{N}}} \wedge \mathtt{N} > 0\right\}$$
```
while(N>1) (
```
$$\left\{\mathtt{X} = k^{2^{i-\mathtt{N}}} \wedge \mathtt{N} > 0 \wedge \mathtt{N} > 1\right\}$$
$$\left\{\mathtt{X} \times \mathtt{X} = k^{2^{i-(\mathtt{N}-1)}} \wedge \mathtt{N} - 1 > 0\right\}$$
```
   X = X*X;
```
$$\left\{\mathtt{X} = k^{2^{i-(\mathtt{N}-1)}} \wedge \mathtt{N} - 1 > 0\right\}$$
```
   N = N-1;
```
$$\left\{\mathtt{X} = k^{2^{i-\mathtt{N}}} \wedge \mathtt{N} > 0\right\}$$
```
)
```
$$\left\{\mathtt{X} = k^{2^{i-\mathtt{N}}} \wedge \mathtt{N} > 0 \wedge \mathtt{N} \leq 1\right\}$$
$$\left\{\mathtt{X} = k^{2^{i-\mathtt{N}}} \wedge \mathtt{N} = 1\right\}$$
$$\left\{\mathtt{X} = k^{2^{i-1}}\right\}$$

Tablica 11.19. Dowód częściowej poprawności programu

11.9. Zadania

Zadanie 11.1. Udowodnij lemat 11.1 i twierdzenie 11.2.

Zadanie 11.2. Udowodnij lematy 11.5, 11.6 i twierdzenie 11.7.

Zadanie 11.3. Udowodnij, że każdy język regularny można opisać przy pomocy wyrażenia z operatorem punktu stałego z podrozdziału 11.6 oraz że język opisywany przez każde takie wyrażenie jest regularny.

Zadanie 11.4. Udowodnij równości (11.4) i (11.6) oraz równoważność semantyki denotacyjnej i operacyjnej wyrażeń regularnych z operatorem punktu stałego z podrozdziału 11.6.

Zadanie 11.5. Udowodnij, że algebra początkowa dla zbioru równości przedstawionych w tablicy 11.13 jest izomorficzna z algebrą zadaną przez semantykę denotacyjną wyrażeń regularnych z tablicy 11.11.

Zadanie 11.6. Za pomocą reguł strukturalnej semantyki operacyjnej opisaliśmy podzbiór języka D nie zawierający instrukcji wejścia/wyjścia. Jeżeli program może się komunikować z otoczeniem, to stan obliczeń nie jest opisany jedynie przez zawartość pamięci, ale

wymaga również uwzględnienia interakcji z otoczeniem. Niech $\mathbb{Z}^0 = \epsilon$, $\mathbb{Z}^{k+1} = \mathbb{Z} \times \mathbb{Z}^k$, $\mathbb{Z}^* = \bigcup_{k=0}^{\infty} \mathbb{Z}^k$ (skończone, być może puste ciągi liczb całkowitych) oraz $\mathbb{Z}^\omega = \mathbb{Z} \times \mathbb{Z}^\omega = \prod_{k=1}^{\infty} \mathbb{Z}$ (nieskończone ciągi liczb całkowitych). Przyjmujemy, że (x, s) oznacza ciąg złożony z liczby x i wszystkich elementów (skończonego lub nie) ciągu s. Każdy niepusty ciąg liczb całkowitych można przedstawić w tej postaci. Stan obliczeń z uwzględnieniem operacji wejścia/wyjścia można opisać za pomocą krotki $\langle \sigma, i, o \rangle \in \mathbb{S} \times \mathbb{Z}^\omega \times \mathbb{Z}^*$, gdzie $\sigma : \mathbb{A} \to \mathbb{Z}$ jest opisem pamięci, $i \in \mathbb{Z}^\omega$ reprezentuje nieskończony ciąg danych wejściowych, zaś $o : \mathbb{Z}^*$ reprezentuje skończony ciąg danych wyjściowych wypisanych przez program. Napisz reguły strukturalnej semantyki operacyjnej języka D z uwzględnieniem operacji wejścia/wyjścia. Zdefiniuj semantykę denotacyjną tego języka.

Zadanie 11.7. Niech $\mathcal{E}$ będzie zbiorem *wyjątków*. Rozszerzamy składnię języka D o instrukcje zgłaszania i obsługi wyjątków:

$$\begin{array}{rcl} \mathcal{C} & ::= & X = \mathcal{A} \mid \mathcal{C} \texttt{ ; } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \texttt{ else } \mathcal{C} \mid \texttt{while (}\mathcal{B}\texttt{) } \mathcal{C} \mid \\ & & \texttt{raise } \mathcal{E} \mid \mathcal{C} \texttt{ handle (}\mathcal{D}\texttt{)} \\ \mathcal{D} & ::= & \texttt{when } \mathcal{E} \texttt{ => } \mathcal{C} \;\; \mathcal{D} \mid \epsilon \\ \epsilon & ::= & \end{array}$$

Niech $\emptyset$ oznacza, że wykonanie instrukcji zakończyło się pomyślnie. Stan obliczeń będzie teraz parą $(\{\emptyset\} \cup \mathcal{E}) \times \mathbb{S}$, gdzie $\mathbb{S}$ jest stanem pamięci. Semantykę instrukcji złożonej opiszemy regułą:

$$\frac{\langle C_1, \sigma \rangle \to \langle \emptyset, \sigma' \rangle \quad \langle C_2, \sigma' \rangle \to \langle E, \sigma'' \rangle}{\langle C_1 C_2, \sigma \rangle \to \langle E, \sigma'' \rangle}$$

zaś propagację wyjątków regułą:

$$\frac{\langle C_1, \sigma \rangle \to \langle E, \sigma' \rangle \quad E \in \mathcal{E}}{\langle C_1 C_2, \sigma \rangle \to \langle E, \sigma' \rangle}$$

Napisz pozostałe reguły strukturalnej semantyki operacyjnej dla nowego języka. Zdefiniuj jego semantykę denotacyjną.

Zadanie 11.8. Oto składnia wyrażeń pewnego języka:

$$\mathcal{A} \quad ::= \quad c \mid X \mid \mathcal{A} + \mathcal{A} \mid \mathcal{A} - \mathcal{A} \mid \mathcal{A} \times \mathcal{A} \mid \mathcal{A} \;/\; \mathcal{A} \mid \mathcal{A} \;\%\; \mathcal{A} \mid \texttt{try } \mathcal{A} \texttt{ otherwise } \mathcal{A}$$

Błąd może wystąpić podczas dzielenia przez zero. Wyrażenie `try` e_1 `otherwise` e_2 ma wartość e_1, jeśli obliczenie e_1 kończy się pomyślnie i e_2, jeśli w trakcie jego obliczenia występuje dzielenie przez zero. Przyjmujemy, że dziedziną interpretacji wyrażeń jest zbiór $\mathbb{D} = \{E\} \cup \mathbb{Z}$ (przy czym $E \notin \mathbb{Z}$). Napisz semantykę operacyjną i denotacyjną tych wyrażeń.

Zadanie 11.9. Wyrażenia arytmetyczne języka D rozszerzamy o operatory postinkrementacji i postdekrementacji:

$$\mathcal{A} \quad ::= \quad c \mid \mathcal{I} \mid -\mathcal{A} \mid \mathcal{A} \oplus \mathcal{A} \mid \mathcal{I}\texttt{++} \mid \mathcal{I}\texttt{--}$$

Teraz więc obliczenie wyrażeń może powodować skutki uboczne. Napisz reguły strukturalnej semantyki operacyjnej dla nowego języka. Zdefiniuj jego semantykę denotacyjną.

Zadanie 11.10. Abstrakcyjna składnia instrukcji języka D++ jest następująca:

$$\begin{aligned}\mathcal{C} \quad ::= \quad & X = \mathcal{A} \mid \mathcal{C} \texttt{ ; } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \texttt{ else } \mathcal{C} \mid \texttt{while (}\mathcal{B}\texttt{) } \mathcal{C} \mid \\ & \texttt{switch (}\mathcal{A}\texttt{) (}\mathcal{D}\texttt{)} \\ \mathcal{D} \quad ::= \quad & \texttt{case } c \texttt{ : } \mathcal{C} \texttt{ ; } \mathcal{D} \mid \epsilon \\ \epsilon \quad ::= \quad & \end{aligned}$$

Semantyka instrukcji

$$\texttt{switch (}e\texttt{) (case } c_1\texttt{: } C_1\texttt{; ... ; case } c_n\texttt{: } C_n\texttt{)}$$

jest następująca: najpierw jest obliczane wyrażenie e, a następnie jego wartości poszukuje się wśród liczb $c_1, \ldots, c_n$. Jeśli istnieją dwie (lub więcej) takie same liczby c_i, wybiera się pierwszą z nich. Gdy już wybrano liczbę c_i, wykonuje się instrukcję C_i. Jeżeli wśród liczb $c_1, \ldots, c_n$ nie ma wartości wyrażenia e, nie wykonuje się żadnej akcji. Napisz gramatykę opisującą konkretną składnię języka D++. Opisz algorytm translacji programów napisanych w języku D++ na programy w języku D (pierwszy kompilator C++ też tak działał!) Narysuj odpowiednie diagramy przepływu. Napisz reguły strukturalnej semantyki operacyjnej dla instrukcji `switch`. Zdefiniuj semantykę denotacyjną języka D++.

Zadanie 11.11. Abstrakcyjna składnia instrukcji języka D−− jest następująca:

$$\mathcal{C} \quad ::= \quad X = \mathcal{A} \mid l \texttt{ : } \mathcal{C} \mid \mathcal{C} \texttt{ ; } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \mid \texttt{goto } l$$

gdzie l jest klasą *etykiet*. Napisz gramatykę opisującą konkretną składnię języka D−−. Opisz algorytm translacji programów napisanych w języku D−− na język D++ i odwrotnie. Narysuj odpowiednie diagramy przepływu. Napisz reguły strukturalnej semantyki operacyjnej języka D−−. Zdefiniuj jego semantykę denotacyjną.

Zadanie 11.12. Język D rozszerzamy o dodatkową konstrukcję `for`, tj. przyjmujemy, że abstrakcyjna składnia instrukcji jest następująca:

$$\begin{aligned}\mathcal{C} \quad ::= \quad & \mathcal{I} = \mathcal{A} \mid \mathcal{C}\,\mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \mid \texttt{if (}\mathcal{B}\texttt{) } \mathcal{C} \texttt{ else } \mathcal{C} \mid \\ & \texttt{while (}\mathcal{B}\texttt{) } \mathcal{C} \mid \texttt{for (}\mathcal{I} \texttt{ from } \mathcal{A} \texttt{ to } \mathcal{A}\texttt{) } \mathcal{C}\end{aligned}$$

Oto trzy różne sposoby zdefiniowania semantyki instrukcji

$$\texttt{for (}X \texttt{ from } e_1 \texttt{ to } e_2\texttt{) } C$$

w której X nazywa się *zmienną sterującą pętli*:

1. Najpierw są obliczane i zapamiętywane wartości k_1 i k_2 wyrażeń e_1 i e_2. Następnie instrukcja C jest wykonywana $k_2 - k_1 + 1$ razy. Przed każdym wykonaniem instrukcji C zmiennej sterującej X jest przypisywana kolejna wartość z przedziału $k_1, \ldots, k_2$. Ewentualne przypisanie zmiennej X innej wartości w instrukcji C nie ma wpływu na liczbę wykonań pętli ani na wartość tej zmiennej w kolejnych wykonaniach instrukcji C.
2. J.w., przy czym teraz zmienna X ma początkowo wartość k_1 i jej wartość jest niejawnie zwiększana o jeden po każdym wykonaniu instrukcji C. Pętlę wykonuje się dopóty, dopóki wartość zmiennej X jest niewiększa niż k_2. Dla przykładu po wykonaniu programu

   ```
   Y = 0;
   for (X from 1 to 12) (
      X = X+1;
      Y = Y+1;
   )
   ```

   zmienna `Y` ma wartość 6 (w poprzednim przypadku miałaby wartość 12).
3. J.w., przy czym teraz wartość wyrażenia e_2 jest wyliczana przed każdym wykonaniem instrukcji C. Pętlę wykonuje się dopóty, dopóki wartość zmiennej X jest niewiększa niż bieżąca wartość wyrażenia e_2. Dla przykładu po wykonaniu programu

   ```
   Z = 12;
   Y = 0;
   for (X from 1 to Z) (
      Z = Z - 1;
      Y = Y+1;
   )
   ```

   zmienna `Y` ma wartość 6.

We wszystkich przypadkach wartość zmiennej sterującej po zakończeniu wykonania pętli jest nieokreślona. Napisz gramatykę opisującą konkretną składnię rozszerzonego języka. Napisz dla każdego przypadku reguły strukturalnej semantyki operacyjnej. Pokaż jak w każdym przypadku dokonać translacji programów w rozszerzonym języku na oryginalny język D. Narysuj odpowiednie diagramy przepływu. Zdefiniuj semantykę denotacyjną we wszystkich trzech przypadkach.

Zadanie 11.13. Oto program w języku D:

```
Y = 1;
while (X != 0) (
   Y = Y*N;
   X = X-1;
)
```

Nazwijmy go P. Korzystając z reguł strukturalnej semantyki operacyjnej udowodnij, że jeśli $\sigma(\mathtt{X}) \geq 0$ i $\langle P, \sigma\rangle \to \sigma'$, to $\sigma'(\mathtt{Y}) = (\sigma(\mathtt{N}))^{\sigma(\mathtt{X})}$. Korzystając z definicji semantyki denotacyjnej udowodnij, że jeśli $\sigma(\mathtt{X}) \geq 0$, to $[\![P]\!]\sigma(\mathtt{Y}) = \sigma(\mathtt{N})^{\sigma(\mathtt{X})}$.

Zadanie 11.14. Wynajdź reguły semantyki aksjomatycznej dla następujących instrukcji:

a) `do` C `while (`b`);`

b) `switch (`e`)` $(n_1 : C_1 ; \ldots ; n_k : C_k)$, gdzie $n_1, \ldots, n_k$ są literałami całkowitoliczbowymi (por. zadanie 11.10);

c) `halt`, która zrywa działanie programu;

d) `for (`X `from` e_1 `to` e_2`)` C, jak w zadaniu 11.12, punkt 1;

e) `for (`X `from` e_1 `downto` e_2`)` C, j.w., tylko wartość X jest zmniejszana o jeden.

Zadanie 11.15. Udowodnij twierdzenie: dla dowolnego programu C zachodzi $\{T\}C\{T\}$. Wyprowadź stąd wniosek: dla dowolnej asercji ϕ i dowolnego programu I zachodzi $\{\phi\}I\{T\}$.

Zadanie 11.16. Napisz drzewo dowodu twierdzenia:

$\{\mathtt{S} = x \cdot y - \mathtt{X} \cdot \mathtt{Y}\}$ `while (!odd(X)) (Y=2*Y; X=X/2)` $\{\mathtt{S} = x \cdot y - \mathtt{X} \cdot \mathtt{Y} \wedge \mathtt{odd(X)}\}$

gdzie

$$\mathtt{odd}(n) = \begin{cases} T, & n \text{ jest nieparzyste} \\ F, & n \text{ jest parzyste} \end{cases}$$

W poniższych zadaniach dopisz pomiędzy instrukcjami asercje tak, by można było z nich utworzyć dowód, że programy spełniają podane specyfikacje.

Zadanie 11.17.

```
{X = x ∧ Y = y}
S = 0;
while (X!=0) (
   while (!odd(X)) (
      Y = 2*Y;
      X = X/2;
   )
    S = S+Y;
    X = X-1;
)
{S = x · y}
```

Zadanie 11.18.

$\{\texttt{X} = x \wedge \texttt{N} = n \wedge n \geq 0\}$
```
Z = 1;
while (N>0) (
   if (odd(N))
      Z = Z*X;
   N = N/2;
   X = X*X;
)
```
$\{\texttt{Z} = x^n\}$

Zadanie 11.19.

$\{\texttt{A} = a \wedge \texttt{B} = b\}$
```
while (A<>B) (
  while (A>B) A = A-B;
  while (B>A) B = B-A;
)
```
$\{\texttt{A} = \texttt{B} \wedge \texttt{A} = \gcd(a, b)\}$

gdzie $\gcd(a, b)$ jest największym wspólnym podzielnikiem liczb a i b.

Zadanie 11.20. Wyznacz wp(`if (X%4!=0) X=X+1`, $\texttt{X} \bmod 2 = 0$).

Zadanie 11.21. Niech

$$C = \texttt{while (Y<=R) (R=R-Y; Q=Q+1)}$$

Wyznacz wp(C, $\texttt{R} < \texttt{Y} \wedge \texttt{X} = \texttt{R} + \texttt{Y} \cdot \texttt{Q}$). (Warunek końcowy oznacza, że `Q` jest ilorazem a `R` resztą z dzielenia `X` przez `Y`.) Dla jakich asercji ϕ jest prawdziwe $\{\phi\}C\{\phi\}$?

Zadanie 11.22. Oto gorliwa wersja operatora alternatywy w C:

```
int or (int x, int y) { return x || y; }
```

Wyjaśnij, czemu `or`(e_1, e_2) jest obliczany gorliwie, zaś e_1 `||` e_2 leniwie. Napisz program, który wypisuje inny wynik, jeśli zamienisz w nim `||` na `or` (tj. pokaż, że strategia wartościowania ma wpływ na znaczenie programu).

Rozdział 12

Specyfikacje algebraiczne

Do uzupełnienia: Idea specyfikacji algebraicznych. Semantyka algebry początkowej. Konstruktory, destruktory, selektory i operatory. Dowody indukcyjne.

Rozdział 13

Programowanie funkcjonalne

Do uzupełnienia:

13.1. Skutki uboczne
13.2. Trwałe i ulotne struktury danych
13.3. Przetwarzanie list w SML-u

Rozdział 14

Algebraiczne typy danych

W rozdziale 12 zajmowaliśmy się opisywaniem semantyki typów danych za pomocą specyfikacji algebraicznych, tj. definiowaniem tzw. *algebraicznych typów danych*. Wiele języków programowania posiada mechanizmy pozwalające na definiowanie takich typów bezpośrednio w programie. Po raz pierwszy algebraiczne typy danych wprowadził Rod Burstall do języka programowania HOPE i języka specyfikacji wykonywalnych CLEAR. W podobnej postaci pojawiły się one następnie w Mirandzie Davida Turnera, Standard ML-u, Haskellu, Concurrent Cleanie i innych językach.

14.1. Algebraiczne typy danych w SML-u

Opisana w paragrafie 9.1.2 deklaracja `type` nie definiuje nowego typu danych ani nowych wartości. Mówimy, że nie jest *generatywna*. W bieżącym podrozdziale opiszemy generatywną deklarację typu, tzw. *deklarację* `datatype`.

14.1.1. Typy wyliczeniowe

W Pascalu możemy definiować typy wyliczeniowe następująco:

```
type color = (Red, Green, Blue);
```

Składnia deklaracji SML-owej jest bardzo podobna:

```
datatype color = Red | Green | Blue
```

Wprowadzono nowy typ danych `color` i trzy jego *konstruktory* o nazwach `Red`, `Green` i `Blue`. Konstruktory są to identyfikatory, za pomocą których możemy budować, *konstruować* wartości danego typu. Wszystkie konstruktory wymienia się w deklaracji `datatype` oddzielając je od siebie kreskami pionowymi. Typ `color` i jego trzy elementy są całkiem nowe, wygenerowane, inne niż cokolwiek do tej pory. W odpowiedzi system wypisuje

```
datatype color = Blue | Green | Red
```

potwierdzając, że `color` jest nazwą nowego typu danych i że `Red`, `Green` i `Blue` są konstruktorami tego typu. Ponadto domyślnie system zdefiniował dwie operacje dla typu `color` — binarne operatory porównania = i <> (zatem `color` jest typem równościowym). Możemy więc napisać

```
- Red = Blue;
val it = false : bool
```

Nie zdefiniowano domyślnie (tak jak np. w Pascalu) operacji `ord`, `pred` i `succ`.[1]

Funkcje działające na wartościach typu `color` moglibyśmy definiować tak jak w Pascalu:

```
fun my_favorite x = x = Green
```

choć wzorce dają dużo więcej swobody, np.

```
fun my_favorite Green = true
  | my_favorite _  = false
```

We wzorcach typu `color` poza zmiennymi mogą pojawiać się jedynie identyfikatory `Red`, `Green` i `Blue`, tylko one są konstruktorami tego typu.

Predefiniowanym typem wyliczeniowym jest:

```
datatype bool = true | false
```

Ze względu na specjalny porządek obliczania, wyrażenia `if`, `andalso` i `orelse` nie mogą być zaprogramowane jako zwyczajne funkcje działające na wartościach typu `bool`.

W podobny sposób możemy też spróbować samodzielnie zdefiniować typ `unit`. Jego konstruktor nie jest niestety identyfikatorem. Gdyby był, to deklaracja typu `unit` wyglądałaby w SML-u następująco:

```
datatype unit = ()
```

14.1.2. Generatywność a widoczność i przesłanianie

Ponieważ deklaracja `datatype` jest *generatywna*, ponowne zadeklarowanie typu o tej samej nazwie lub powtórne użycie konstruktora istniejącego typu w deklaracji nowego typu powoduje przesłonięcie poprzednich typów i konstruktorów. Przestają być one dostępne na zawsze (zatem działa tu zasada wiązania statycznego), np.

```
datatype color = Red | Green | Blue
fun my_favorite Green = true
  | my_favorite _  = false
```

[1] Deklaracja `datatype`, o czym przekonamy się za chwilę, jest dużo ogólniejsza od Pascalowej, te operacje nie we wszystkich przypadkach dałoby się zdefiniować sensownie. W Mirandzie Turner wprowadził pewien porządek na konstruktorach, jednak takie rozwiązanie nie jest ani eleganckie, ani przydatne.

definiuje nowy typ `color` i funkcję `my_favorite : color -> bool`. Jeżeli powtórnie zadeklarujemy typ `color`:

```
datatype color = Red | Green | Blue
```

wówczas wywołanie funkcji `my_favorite` spowoduje błąd:

```
- my_favorite Red;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: ?.color
  operand:         color
  in expression:
    my_favorite Blue
```

Funkcja `my_favorite` odwołuje się do „starego" typu `color`, którego nazwa jest niedostępna (dlatego system pisze `?.color`, kompilator edynburski pisał `color[hidden]`), podobnie jak i wartości. Konstruktor `Red` po powtórnej deklaracji jest elementem nowego typu. Funkcja `my_favorite` jest bezużyteczna, ponieważ nie zachowaliśmy żadnej wartości pasującej do typu jej argumentu. Często powyższe zjawisko ma miejsce, gdy poprawiamy i wielokrotnie kompilujemy program. Powtórna definicja typu wymaga skompilowania wszystkich funkcji działających na jego wartościach.

Jeżeli definicji typu używamy jedynie lokalnie, warto zanurzyć ją w deklaracji `local` lub wyrażeniu `let`. Zasięg obowiązywania takiej deklaracji jest ograniczony tymi samymi regułami, co zasięg obowiązywania deklaracji wartości. Ujemną stroną takiej możliwości jest niebezpieczeństwo, że wartości typu „przeżyją" jego nazwę (podobne zjawisko widzieliśmy już przy deklaracjach wyjątków), np.

```
- local
=   datatype me = Me
= in
=   val x = Me
= end;
val x = Me : ?.me
```

Ani nazwa, ani konstruktory typu wartości `x` nie są już dostępne. Nie jest to zbyt eleganckie, ale nie prowadzi do żadnych komplikacji, dlatego system SML/NJ nie protestuje przed skompilowaniem powyższej deklaracji. Natomiast standard języka zabrania, by typ w jakikolwiek sposób opuścił zasięg swojej deklaracji.

14.1.3. Konstruktory funkcyjne

SML-owa deklaracja `datatype` pozwala definiować znacznie ogólniejsze od typów wyliczeniowych, nawet nieskończone kolekcje danych. Możliwe jest to dzięki temu, że konstruktory wartości definiowanego mogą być nie tylko stałymi, lecz także funkcjami (podobnie jak wyjątki z parametrami). Wystarczy po nazwie konstruktora w jego deklaracji napisać słowo kluczowe `of` i wyrażenie typowe opisujące typ jego argumentu (typu wartości nie ma po co pisać, bo jest nim zawsze typ właśnie definiowany), np.

```
datatype number = Num of int | Infty
```

Powyższa deklaracja wprowadza nowy typ `number` i dwa jego konstruktory:

```
Infty : number
  Num : int -> number
```

Konstruktor `Infty`, podobnie jak w typach wyliczeniowych, jest po prostu nazwą wartości typu `number`, natomiast `Num` jest funkcją, maszyną do tworzenia różnych wartości typu `number`. Jest ich tyle, ile danych typu `int`. Wszystkie są parami różne i są innego typu niż `int`. Zatem fragment deklaracji `datatype` postaci

$$\texttt{datatype}\ T\ \texttt{=}\ \ldots\ \texttt{|}\ C\ \texttt{|}\ \ldots$$

wprowadza jedną wartość $C : T$, podczas gdy

$$\texttt{datatype}\ T\ \texttt{=}\ \ldots\ \texttt{|}\ C\ \texttt{of}\ \sigma\ \texttt{|}\ \ldots$$

wprowadza funkcję $C : \sigma \to T$ do konstruowania wartości typu T. Zaaplikowanie konstruktora `Num` do liczby całkowitej tworzy wartość typu `number`, np. `Num 5`, `Num 7` itd., są typu `number`.

Typ T zdefiniowany deklaracją postaci

$$\texttt{datatype}\ T\ \texttt{=}\ F_1\ \texttt{of}\ \sigma_1\ \texttt{|}\ \ldots\ \texttt{|}\ F_n\ \texttt{of}\ \sigma_n\ \texttt{|}\ C_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_m$$

jest równościowy dokładnie wtedy, gdy wszystkie typy σ_i, dla $i = 1, \ldots, n$, są równościowe. O ile w przypadku typów wyliczeniowych można by (jeśli ktoś jest masochistą) żmudnie używać kaskady wyrażeń `if` i relacji `=` do zbadania, z jaką wartością mamy do czynienia, o tyle dla bardziej skomplikowanych typów, takich jak `number` z ostatniego przykładu, wzorce są praktycznie jedynym mechanizmem dostępu do danych zdefiniowanych deklaracją `datatype`. Możemy dla przykładu zdefiniować funkcję `++` dodającą wartości typu `number`:

```
infix 6 ++
fun (Num n) ++ (Num m) = (Num (m+n) handle Overflow => Infty)
  |        _ ++ _       = Infty
```

Ponieważ konstruktor `Num` jest funkcją, we wzorcu powinien być zaaplikowany do innego wzorca, reprezentującego jego argument (np. zmiennej).

14.1.4. Jak uchronić się przed nadwagą

Generatywność deklaracji `datatype` możemy wykorzystać do tego, by zmusić system typów do kontroli poprawności użycia danych. Deklaracja `type` nie daje takiej możliwości. Rozważmy bowiem obliczenia antropometryczne dotyczące wzrostu i wagi osób. Aby jawnie wskazać, które dane reprezentują wzrost, a które wagę (obie są liczbami typu `real`, zwykle nieujemnymi), możemy wprowadzić dwa *synonimy* typu `real` z pomocą deklaracji `type` i zdefiniować odpowiednie stałe, np.

```
type height = real
 and weight = real
val h : height = 170.0
and w : weight = 60.0
```

System pamięta o naszych typach i na pytanie, ile bym mierzył, gdybym był wyższy o 10%, sensownie odpowiada:

```
- h + 0.1 * h;
val it = 187.0 : height
```

Niestety radość programisty pryska z chwilą kompilacji poniższego wyrażenia:

```
- h + w;
val it = 230.0 : height
```

System pozwala dodać wzrost do wagi! Faktycznie nie ma żadnej kontroli typów! Możemy temu zapobiec z pomocą deklaracji `datatype`:

```
datatype height = Height of real
     and weight = Weight of real
val h = Height 170.0
and w = Weight  60.0
```

Obecnie wzrost i waga są osobnymi typami danych, a ich konstruktory, to jakby funkcje jawnie zmieniające typ. Teraz już nie można napisać

```
- h+w;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: height * height
  operand:         height * weight
  in expression:
    h + w
Error: overloaded variable not defined at type
  symbol: +
  type: height
```

ponieważ + nie jest zdefiniowany dla tych typów. Możemy jednak zdefiniować dla nich własne operacje dodawania:

```
fun addHeight (Height x, Height y) = Height (x+y)
fun addWeight (Weight x, Weight y) = Weight (x+y)
```

System pozwoli dodać np. wzrost do wzrostu (ile bym mierzył, gdybym stanął sobie na głowie):

```
- addHeight(h,h);
val it = Height 340.0 : height
```

ale dodanie wzrostu do wagi jest wykluczone:

```
- addHeight(h,w);
Error: operator and operand don't agree [tycon mismatch]
  operator domain: height * height
  operand:         height * weight
  in expression:
    addHeight (h,w)
```

Możecie protestować, że jest to zbyt restrykcyjne, niekiedy są bowiem sytuacje w których dodanie wzrostu do wagi jest sensowne, np. we wzorze na nadwagę! Istotnie, w tym przypadku należy od wzrostu odjąć jeden metr i pomniejszyć to co wyjdzie o 10%. Jeżeli wynik jest mniejszy niż nasza waga — mamy nadwagę! Porównujemy więc tu wagę ze wzrostem i prawa układu jednostek miar są pogwałcone. Możemy zaprogramować i taki predykat, należy jedynie jawnie pokazać we wzorcach, gdzie używamy wzrostu, a gdzie wagi:

```
- fun overweight (Height h, Weight w) = 0.9 * (h - 100.0) < w;
val overweight = fn : height * weight -> bool
- overweight (h,w);
val it = false : bool
```

i komputer stwierdza, że nie mam nadwagi! Zauważmy że system dla funkcji `overweight` wywnioskował typ `height * weight -> bool` i zamiast dwóch anonimowych liczb rzeczywistych żąda — w odpowiedniej kolejności — wzrostu i wagi. Dzięki temu program jest dużo bardziej odporny na pomyłki. Gdyby funkcję `overweight` zaprogramować wprost na liczbach rzeczywistych:

```
- fun overweight (h,w) = 0.9 * (h - 100.0) < w;
val overweight = fn : real * real -> bool
```

wówczas drobna i trudna do wykrycia pomyłka:

```
- overweight (60.0,170.0);
val it = true : bool
```

mogłaby mnie wprawić w niepotrzebne kompleksy (a swoją drogą, wyobrażacie sobie człowieka wzrostu 60 cm ważącego 170 kilogramów?!). Z punktu widzenia efektywności obliczeń przetwarzanie danych typu `height` postaci `Height h` jest niemal tak efektywne, jak operowanie samą liczbą rzeczywistą `h`. Wygoda i bezpieczeństwo, jakie uzyskuje się wprowadzając osobny typ danych znacznie przewyższa te koszty.

14.1.5. Typy polimorficzne

Jeżeli w wyrażeniu typowym opisującym typ konstruktora pojawi się zmienna typowa, należy ją wymienić jako parametr typu w postaci podobnej, jak to czynimy w deklaracji `type`:

$$\texttt{datatype}\ \vec{\alpha}\ T\ \texttt{=}\ \ldots$$

gdzie $\vec{\alpha}$ jest albo pustym napisem (wówczas typ jest monomorficzny i nie ma parametrów), pojedynczą zmienną typową, lub listą zmiennych typowych oddzielonych przecinkami i ujętych w nawiasy `()`, np.

```
datatype ('a,'b) pair = Pair of 'a * 'b
```

Wymaganie, by każda zmienna typowa została wymieniona jako parametr typu jest istotne i sprawdzane skrupulatnie przez kompilator. Bez niego system typów załamałby się. Rozważmy dla przykładu deklarację

```
datatype T = C of 'a
fun conv x = case C x of C x => x
```

Ponieważ `C : 'a -> T`, to `C x : T`. Funkcja `conv`, choć jest w istocie identycznością, ma typ `'a -> 'b` i z jej pomocą możemy np. dodać `true` do jedynki: `conv true + 1`.[2] Z tych powodów SML nie przyjmuje powyższej deklaracji typu pisząc:

```
Error: unbound type variable in type declaration: 'a
```

W standardowym środowisku SML/NJ jest dostępny niezwykle użyteczny polimorficzny typ

```
datatype 'a option = SOME of 'a | NONE
```

Z jego pomocą możemy bez użycia wyjątków zdefiniować funkcję, której obliczenie *niekiedy* dostarcza wartości, niekiedy zaś nie. Np. funkcja ujawniająca głowę listy może mieć typ `'a list -> 'a option` i być zdefiniowana następująco:

```
fun hd (x::_) = SOME x
  | hd nil = NONE
```

Teraz jednak *każde* wywołanie funkcji `hd` będzie się wiązać z koniecznością sprawdzenia (np. z pomocą wyrażenia `case`) który z wariantów faktycznie zaszedł. Jeśli przewidujemy, że wynik będziemy badać po każdym wywołaniu funkcji, dużo bardziej elegancko jest użyć typu `'a option`, w przeciwnym razie lepiej korzystać z wyjątków.

14.1.6. Typy rekurencyjne

W wyrażeniu typowym opisującym typ argumentu konstruktora może wystąpić nazwa definiowanego typu danych. Dzięki temu definiowane przez nas typy danych mogą mieć rekurencyjną strukturę. Możemy np. sami zdefiniować listy w takiej samej postaci, w jakiej predefiniowano je w systemie:

```
infixr 5 ::
datatype 'a list = :: of 'a * 'a list | nil
```

[2] Jest do pomyślenia rozszerzenie deklaracji `datatype`, w której opuszczenie zmiennej typowej powodowałoby jej egzystencjalną kwantyfikację. Typem `C x` byłby wówczas $\exists\alpha.(\alpha\ \mathtt{T})$ i powyższe oszustwo nie byłoby możliwe. Takie rozwiązanie jest zaimplementowane np. w języku Concurrent Clean.

Rekurencyjne definicje typów pojawiają się często tam, gdzie występują naturalnie rekurencyjne zależności między danymi, np. w parserach. Rozważmy następującą gramatykę abstrakcyjną wyrażeń:

⟨wyrażenie⟩ ::= ⟨liczba⟩ | ⟨wyrażenie⟩ + ⟨wyrażenie⟩ | ⟨wyrażenie⟩ ∗ ⟨wyrażenie⟩

Abstrakcyjne drzewa rozbioru wyrażeń możemy reprezentować jako dane typu

```
infix 6 `+`
infix 7 `*`
datatype wyrazenie =  $  of int |
                     `+` of wyrazenie * wyrazenie |
                     `*` of wyrazenie * wyrazenie
```

Parser wyrażeń będzie funkcją typu `string -> wyrazenie` i będzie w wyniku dostarczał abstrakcyjne drzewa rozbioru tych wyrażeń. Abstrakcyjne drzewo rozbioru wyrażenia `3*(12+5)+7` zapiszemy w SML-u w postaci `$3 `*` ($12 `+` $5) `+` $7`.

Deklaracje `datatype` można też łączyć słowem `and`. Dzieki temu możemy definiować typy *wzajemnie* rekurencyjne, np.

```
datatype book   = Book of {title : string, read_by : reader list}
     and reader = Reader of {name : string, borrowed : book list}
```

W karcie bibliotecznej (wartość typu `book`) jest zapisany tytuł książki i lista czytelników, którzy dotychczas daną książkę pożyczyli. Karta czytelnika (typu `reader`) zawiera jego imię i listę aktualnie wypożyczonych książek. Typy `book` i `reader` wzajemnie się do siebie odwołują.
Rozważmy (trochę dziwaczną) funkcję

```
fun pairList (x1::x2::xs) = (x1,x2) :: pairList xs
  | pairList nil = nil
```

Kompilator protestuje, że

```
Warning: match nonexhaustive
          x1 :: x2 :: xs => ...
          nil => ...
```

i ma rację, ponieważ wywołanie `pairlist [1,2,3]` kończy się katastrofą. Chcielibyśmy, by system typów kontrolował w jakiś sposób, czy lista ma parzystą liczbę elementów, czy nie. Możemy w tym celu zdefiniować dwa typy danych: osobno list o parzystej i nieparzystej liczbie elementów:

```
infixr 5 / \
datatype 'a evenlist = \ of 'a * 'a oddlist | %
     and 'a oddlist  = / of 'a * 'a evenlist
```

Listę `[1,2,3]` zapiszemy jako wyrażenie `1 / 2 \ 3 / %` typu `int oddlist`, zaś czteroelementowa lista `1 \ 2 / 3 \ 4 / %` jest typu `int evenlist`. Implementacja funkcji `pairlist` dla nowego typu list będzie typu `'a evenlist -> ('a * 'a) list` (wynikiem jest `list`-a, ponieważ nie wiemy, czy będzie miała parzystą, czy nieparzystą liczbę elementów):

```
fun pairlist (x1 \ x2 / xs) = (x1,x2) :: pairlist xs
  | pairlist % = nil
```

Kompilator obecnie nie protestuje, ponieważ wzorce *są* wyczerpujące — system typów gwarantuje, że `pairlist` nie zostanie zaaplikowana do listy o nieparzystej liczbie elementów.

Najbardziej elegancko byłoby, gdyby typy `'a evenlist` i `'a oddlist` były *podtypami* typu `'a list`. Systemy typów z podtypami są jednak trudnym zagadnieniem i wymagają dalszych badań zanim zostaną zaimplementowane w opisanej wyżej postaci.

Rekurencyjną definicję typu można też wykorzystać do opisu całego rodzaju ludzkiego (w sensie biblijnym). Otóż pierwszymi ludźmi byli Adam i Ewa. Człowiek, to:

- Adam,
- Ewa,
- k-ty potomek dwóch innych ludzi, dla $k \geq 0$ (przyjmijmy, że numerujemy od zera).

Otrzymujemy natychmiast SML-ową definicję typu

```
datatype czlowiek = Adam |
                    Ewa |
                    Potomek of word * czlowiek * czlowiek
```

Każdemu człowiekowi odpowiada pewien element typu `czlowiek` oraz dwóm różnym ludziom odpowiadają różne wartości tego typu. Niestety pewne wartości typu `czlowiek` nawet potencjalnie nie mogą odpowiadać żadnemu człowiekowi, np.

```
Potomek(0w5, Adam, Adam)
```

Możemy jednak poprawić naszą definicję. Mężczyzna, to:

- Adam,
- k-ty syn mężczyzny i kobiety,

a kobieta, to

- Ewa,
- k-ta córka mężczyzny i kobiety.

Wniosek: ludzie powstają w wyniku rekursji wzajemnej. W SML-u możemy ten fakt wyrazić następująco:

```
datatype mezczyzna = Adam | Syn   of word * mezczyzna * kobieta
     and kobieta   = Ewa  | Corka of word * mezczyzna * kobieta
```

Zachęcam Czytelnika, by spróbował napisać wartość, która jest jego nazwą.

$$
\begin{array}{lcl}
\texttt{datatype}\ \vec{\alpha}_1\ T_1 & = & F_1^1\ \texttt{of}\ \sigma_1^1 \mid \ldots \mid F_{n_1}^1\ \texttt{of}\ \sigma_{n_1}^1 \mid C_1^1 \mid \ldots \mid C_{m_1}^1 \\
\quad \texttt{and}\ \vec{\alpha}_2\ T_2 & = & F_1^2\ \texttt{of}\ \sigma_1^2 \mid \ldots \mid F_{n_2}^2\ \texttt{of}\ \sigma_{n_2}^2 \mid C_1^2 \mid \ldots \mid C_{m_2}^2 \\
 & \vdots & \\
\quad \texttt{and}\ \vec{\alpha}_k\ T_k & = & F_1^k\ \texttt{of}\ \sigma_1^k \mid \ldots \mid F_{n_k}^k\ \texttt{of}\ \sigma_{n_k}^k \mid C_1^k \mid \ldots \mid C_{m_k}^k \\
\texttt{withtype}\ \vec{\beta}_1\ S_1 & = & \tau_1 \\
\quad \texttt{and}\ \vec{\beta}_2\ S_2 & = & \tau_2 \\
 & \vdots & \\
\quad \texttt{and}\ \vec{\beta}_l\ S_l & = & \tau_l
\end{array}
$$

Tablica 14.1. Postać deklaracji `datatype` w SML-u

14.1.7. Deklaracja withtype

Ciąg wzajemnie rekurencyjnych deklaracji `datatype` możemy połączyć słowem `and`. Deklaracje `type` choć nierekurencyjne, również można łączyć słowem `and`. Jednak `datatype` i `type` to różne deklaracje, dlatego ich ze sobą nie można mieszać. Niekiedy jest to kłopotliwe. Najsłynniejszy przykład pochodzi od Robina Milnera:

```
datatype 'a tree = Node of 'a forest
type 'a forest = 'a tree list
```

Definicja typu `tree` odwołuje się do typu `forest` i odwrotnie. Aby móc łączyć oba rodzaje deklaracji, wprowadzono specjalne słowo kluczowe `withtype` które zastępuje słowo `type`. Dzięki niemu możemy połączyć deklarację `datatype` z deklaracją `type`:

```
- datatype 'a tree = Node of 'a forest
= withtype 'a forest = 'a tree list;
datatype 'a tree = Node of 'a tree list
type 'a forest = 'a tree list
```

14.1.8. Semantyka deklaracji datatype

Najogólniejsza postać deklaracji `datatype` jest przedstawiona w tablicy 14.1. Zbiory elementów nowo utworzonych typów $\vec{\rho}\ T_i$ dla dla dowolnych typów $\vec{\rho}$ oraz $i = 1, \ldots, k$ są opisane następującą definicją rekurencyjną:

1. wartość C_j^i jest elementem typu $\vec{\rho}\ T_i$, dla $j = 1, \ldots, m_i$;
2. jeżeli E jest wartością typu $\sigma_j^i[\vec{\alpha}/\vec{\rho}]$, to $F_j^i E$ jest elementem typu $\vec{\rho}\ T_i$, dla $j = 1, \ldots, n_i$;
3. zbiór elementów typu $\vec{\rho}\ T$ jest najmniejszym zbiorem spełniającym własności 1–2.

Za pomocą definicji rekurencyjnej możemy też opisać zbiór wszystkich wzorców typu $\vec{\rho}\ T$. Będziemy mówić, że x jest identyfikatorem *prefiksowym*, jeśli jest identyfikatorem i nie został zadeklarowany jako infiksowy, lub jest napisem postaci `op` y, gdzie y jest identyfikatorem infiksowym. Identyfikator x jest *zmienną*, jeśli nie jest *konstruktorem* żadnego typu.

1. Prefiksowy konstruktor C_j^i jest wzorcem typu $\vec{\rho}\ T_i$, dla $j = 1, \ldots, m_i$;
2. jeżeli P jest wzorcem typu $\sigma_j^i[\vec{\alpha}/\vec{\rho}]$, a konstruktor funkcyjny F_j^i jest prefiksowy, to napis $F_j^i\,P$ jest wzorcem typu $\vec{\rho}\ T_i$, dla $j = 1, \ldots, n_i$; aplikacja, jak wszędzie, wiąże najsilniej, jeśli więc P jest złożonym wyrażeniem, należy je ująć w nawiasy;
3. jeżeli P_1 i P_2 są wzorcami typów odpowiednio ρ_1 i ρ_2, gdzie $\sigma_j^i[\vec{\alpha}/\vec{\rho}] = \rho_1 * \rho_2$, a konstruktor funkcyjny F_j^i jest infiksowy, to $P_1\ F_j^i\ P_2$ jest wzorcem typu $\vec{\rho}\ T_i$;
4. jeżeli P jest wzorcem typu $\vec{\rho}\ T_i$, to (P) jest wzorcem typu $\vec{\rho}\ T_i$;
5. jeżeli prefiksowy identyfikator x jest zmienną, to x jest wzorcem typu $\vec{\rho}\ T_i$;
6. jeżeli prefiksowy identyfikator x jest zmienną, a P jest wzorcem typu $\vec{\rho}\ T_i$, to x `as` P jest wzorcem typu $\vec{\rho}\ T_i$. Słowo `as` wiąże słabiej niż aplikacja konstruktora do argumentu, dlatego we wzorcu postaci F (x `as` P) nawiasy są niezbędne;
7. znak podkreślenia „_” jest wzorcem typu $\vec{\rho}\ T_i$;
8. zbiory wzorców typu $\vec{\rho}\ T_i$ są najmniejszymi zbiorami spełniającymi warunki 1–7.

Nadto aby wzorzec mógł być użyty w programie SML-owym musi być liniowy, tj. każda zmienna może w nim wystąpić co najwyżej raz.

14.1.9. Wyjątki jako typ danych

W początkowym okresie rozwoju SML-a wyjątki były traktowane jako specjalne klasy obiektów. Nie były wartościami. W roku 1988 Robin Milner zaproponował, by utworzyć specjalny typ danych `exn` i by wyjątki były jego konstruktorami. Jedyna różnica w porównaniu z innymi `datatype`-mi polega na tym, że możemy definiować nowe konstruktory już istniejącego typu, dlatego mówimy, że `exn` jest (jedynym w SML-u) *typem otwartym*. Deklarowanie wyjątków takie samo znaczenie, jak gdyby zdefiniowano wyjątki standardowe deklaracją

```
datatype exn = Match | Bind
```

a następnie każda deklaracja wyjątku postaci

`exception` E lub `exception` E `of` σ

dodawała do powyższej deklaracji `datatype` nowy fragment postaci

$\ldots$ `|` E `|` $\ldots$ lub $\ldots$ `|` E `of` σ `|` $\ldots$

Zastanawiano się, czy nie uczynić otwartości ogólną cechą wszystkich `datatype`-ów w SML-u. Z powodu kłopotów z polimorfizmem (por. definiowanie polimorficznych wyjątków) nie wprowadzono takiego rozszerzenia do standardu SML-a. Drugą przyczyną jest wątpliwość, czy takie typy byłyby użyteczne.

Typ parametru wyjątku powinien być monomorficzny. Gdyby dopuścić, by typ wyjątku mógł zawierać zmienne typowe, to system typów nie działałby poprawnie. Rozważmy bowiem program:

```
let
  val (E,f) =
    let
      exception E of 'a
    in
      (E, fn E x => x)
    end
in
  f (E true) + 1
end
```

Skoro `E` jest typu $\alpha \to$ `exn`, więc w szczególności `bool` $\to$ `exn`, to `E true` ma typ `exn` i może być argumentem funkcji `f`. Z drugiej strony `f` : `exn` $\to \alpha$, więc w szczególności `exn` $\to$ `int` i jej wynik może być dodany do jedynki. W efekcie system typów dopuściłby dodanie wartości logicznej do liczby całkowitej! Problem polega na tym, że w deklaracji `datatype` wszystkie zmienne typowe występujące w typach parametrów konstruktorów powinny być wymienione na liście parametrów typu. Tymczasem typ `exn` nie ma parametrów. Jest to jedna z komplikacji wynikających z tego, że typ `exn` jest definiowany „na raty”, tj. jest typem otwartym.

Z podobnych powodów typ `exn` nie jest typem równościowym. Z definicji typ wprowadzony deklaracją `datatype` jest równościowy, gdy równościowe są wszystkie typy parametrów jego konstruktorów. W przypadku typu otwartego nie można tej własności ustalić *a priori*.

14.2. Drzewa

Drzewo to albo *liść*, albo obiekt zbudowany z *wierzchołka wewnętrznego*, zwanego *korzeniem* i (być może nawet nieskończonej liczby) mniejszych drzew, zwanych jego poddrzewami. Często spotyka się *drzewa binarne*, w których każdy wierzchołek wewnętrzny ma dokładnie dwa poddrzewa. Będziemy rozważać drzewa, w których kolejność poddrzew jest istotna, tj. poddrzewa korzenia będą ustawione w ciąg. Będziemy też zakładać, że poddrzew jest skończona liczba. Formalnie zbiór drzew możemy zadać definicją rekurencyjną. Niech L będzie zbiorem etykiet liści, a W zbiorem etykiet wierzchołków wewnętrznych. Zakładamy, że oba zbiory są niepuste.

1. Każdy liść $l \in L$ jest drzewem.

2. Jeżeli $t_1, \ldots, t_n$ są drzewami, a $w \in W$ — wierzchołkiem wewnętrznym, to

$$\langle w, \langle t_1, \ldots, t_n \rangle \rangle$$

jest drzewem, którego korzeniem jest w i które ma n poddrzew $t_1, \ldots, t_n$, $n \geq 1$.

3. Wszystkie drzewa można zbudować w skończonej liczbie kroków w podany wyżej sposób.

14.2.1. Podstawowe definicje i własności drzew

Zbiór wierzchołków wewnętrznych $W(t)$ drzewa t, to zbiór złożony z etykiety jego korzenia i ze zbioru wierzchołków wewnętrznych wszystkich jego poddrzew. Zbiór wierzchołków wewnętrznych pojedynczego liścia jest pusty:

$$\begin{aligned} W(l) &= \emptyset, \qquad l \in L \\ W(\langle w, \langle t_1, \ldots, t_n \rangle \rangle) &= \{w\} \cup \bigcup_{i=1}^{n} W(t_i), \qquad w \in W \end{aligned}$$

Analogicznie definiujemy *zbiór* $L(t)$ *liści drzewa* t:

$$\begin{aligned} L(l) &= \{l\}, \qquad l \in L \\ L(\langle w, \langle t_1, \ldots, t_n \rangle \rangle) &= \bigcup_{i=1}^{n} L(t_i), \qquad w \in W \end{aligned}$$

Ścieżką w drzewie nazywamy ciąg etykiet jego wierzchołków, taki, że

1. ciąg pusty ε jest ścieżką w drzewie l, dla $l \in L$;
2. jeśli $\langle w_1, \ldots, w_n \rangle$ jest ścieżką w drzewie t_i, $i = 1, \ldots, n$, to ciąg $\langle w, w_1, \ldots, w_n \rangle$ jest ścieżką w drzewie $\langle w, \langle t_1, \ldots, t_n \rangle \rangle$, dla $w \in W$;
3. nie ma innych ścieżek w drzewach poza opisanymi wyżej.

Wysokością drzewa będziemy nazywać długość najdłuższej ścieżki w tym drzewie.

Korzystając wprost z definicji drzew łatwo udowodnić:

Twierdzenie 14.1 (Zasada indukcji dla drzew). Jeżeli własność Φ:

1. jest spełniona dla każdego liścia $l \in L$;
2. założenie że Φ jest spełniona dla poddrzew $t_1, \ldots, t_n$ pociąga prawdziwość Φ dla drzewa $\langle w, \langle t_1, \ldots, t_n \rangle \rangle$ dla każdego wierzchołka wewnętrznego $w \in W$,

wówczas własność Φ jest spełniona dla wszystkich drzew.

Rekursja strukturalna to metoda definiowania funkcji działających na drzewach, w której schemat rekursji odpowiada strukturze drzewa. Wartość funkcji f, zdefiniowanej z pomocą rekursji strukturalnej, dla liścia l jest pewną stałą c_l.[3] Aby obliczyć wartość funkcji f dla drzewa $\langle w, \langle t_1, \ldots, t_n\rangle\rangle$, należy wpierw obliczyć jej wartości $f(t_1), \ldots, f(t_n)$ dla jego poddrzew, a następnie zaaplikować do nich i oryginalnych poddrzew specjalną funkcję g_w, dobraną odpowiednio dla wierzchołka w:

$$\begin{aligned} f(l) &= c_l, \quad l \in L \\ f(\langle w, \langle t_1, \ldots, t_n\rangle\rangle) &= g_w(t_1, \ldots, t_n, f(t_1), \ldots, f(t_n)), \quad w \in W \end{aligned}$$

Schemat iteracji otrzymamy przez odrzucenie parametrów $t_1, \ldots, t_n$ funkcji g_w:

$$\begin{aligned} f(l) &= c_l, \quad l \in L \\ f(\langle w, \langle t_1, \ldots, t_n\rangle\rangle) &= g_w(f(t_1), \ldots, f(t_n)), \quad w \in W \end{aligned}$$

W większości przypadków (jeśli tylko wierzchołek w ma więcej niż jedno poddrzewo) rekursja strukturalna dla drzew nie jest liniowa. Zamiast kolekcji stałych c_l, $l \in L$, będziemy niekiedy mówić o funkcji c określonej na zbiorze L. Podobnie zamiast wielu funkcji g_w będziemy używać jednej funkcji g otrzymującej jako dodatkowy parametr wierzchołek $w \in W$.

Obejść drzewo binarne, znaczy *odwiedzić* wszystkie wierzchołki wewnętrzne i/lub liście drzewa w ustalonym porządku:

infiksowo, gdy wierzchołki lewego poddrzewa są wizytowane jako pierwsze, następnie korzeń, a na koniec wierzchołki prawego poddrzewa;

prefiksowo, gdy na początek jest wizytowany korzeń, potem wierzchołki lewego poddrzewa, a na koniec wierzchołki prawego poddrzewa;

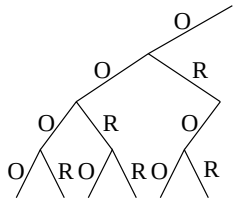
postfiksowo, gdy wierzchołki lewego poddrzewa są wizytowane jako pierwsze, następnie wierzchołki prawego poddrzewa, a na koniec korzeń.

Odwiedzenie polega na wykonaniu pewnej czynności dla wierzchołka/liścia drzewa.

14.2.2. Implementacja drzew w SML-u

Do zaprogramowania drzew w SML-u użyjemy deklaracji `datatype`. Najpierw rozważymy drzewa binarne. Musimy ustalić, czym są zbiory etykiet wierzchołków wewnętrznych i liści drzewa. W najprostszej wersji oba zbiory są jednoelementowe. Pomimo prostoty, takie drzewa są bardzo przydatne. Rozważmy na przykład gry polegające na rzucaniu monetą. Lewe poddrzewo będzie symbolizować wyrzucenie orła, prawe — reszki. Dowolne drzewo jest wówczas zbiorem pewnych ciągów losowań. Niech zwycięstwo polega na tym, że w trakcie rzucania monetą wyniki dotychczasowych losowań zawierają co najmniej tyle orłów co reszek. Drzewo reprezentujące wygrywające ciągi czterech losowań w tej grze jest przedstawione na rysunku 14.1, na którym zaznaczono jedynie wierzchołki wewnętrzne drzewa. Drzewo jest więc w tym przypadku albo pojedynczym liściem (nazwijmy go `Leaf`), albo wierzchołkiem wewnętrznym (niech nazywa się `Node`) związanym z dwoma poddrzewami:

[3] Stała c_l może być dowolną wartością, w szczególności funkcją.



Rysunek 14.1. Drzewo wygrywających ciągów czterech rzutów monetą w pewnej grze

```
datatype tree = Node of {O : tree, R : tree} | Leaf
```

Drzewo z rysunku 14.1 jest wartością

```
Node{O = Node{O = Node{O = Node{O = Leaf,
                                R = Leaf},
                       R = Node{O = Leaf,
                                R = Leaf}},
              R = Node{O = Node{O = Node{O = Leaf,
                                         R = Leaf},
                                R = Leaf},
                       R = Leaf}},
     R = Leaf}
```

typu `tree`. Ponieważ taki napis nie jest zbyt czytelny, lepiej zdefiniować wierzchołek wewnętrzny jako operator infiksowy i użyć identyfikatorów symbolicznych:

```
datatype tree = /\ of tree * tree | L
infix 4 /\
```

Operator `/\` łączy w lewo, można więc opuszczać niektóre nawiasy. Drzewo z rysunku 14.1 będzie teraz reprezentowane przez wyrażenie

```
L/\L /\ (L/\L) /\ (L/\L/\L/\L) /\ L
```

typu `tree`.

Zasada indukcji dla typu `tree` upraszcza się do postaci:

$$\Phi(\mathtt{L}) \ \wedge \ (\forall\, t_1, t_2 : \mathtt{tree}.\ \Phi(t_1) \wedge \Phi(t_2) \Rightarrow \Phi(t_1 \ \mathtt{/\backslash}\ t_2)) \quad \Longrightarrow \quad \forall\, t : \mathtt{tree}.\ \Phi(t)$$

Z jej pomocą możemy dla przykładu udowodnić, że liczba liści drzewa (oznaczmy ją $|\cdot|_L$) jest co najwyżej o jeden większa od liczby wierzchołków wewnętrznych (którą będziemy oznaczać $|\cdot|_W$) tego drzewa. Formalnie obie wielkości należy zdefiniować z pomocą rekursji:

$$\begin{aligned} |\mathtt{L}|_L &= 1 \\ |t_1 \ \mathtt{/\backslash}\ t_2|_L &= |t_1|_L + |t_2|_L \\ |\mathtt{L}|_W &= 0 \\ |t_1 \ \mathtt{/\backslash}\ t_2|_W &= |t_1|_W + |t_2|_W + 1 \end{aligned}$$

(zauważmy, że jest to rekursja strukturalna) i wówczas nasze twierdzenie przyjmie postać:

$$\forall\, t : \texttt{tree}.\ \ |t|_L \leq |t|_W + 1$$

Istotnie, dla drzewa złożonego z pojedynczego liścia:

$$|\mathrm{L}|_L = 1 = 0 + 1 = |\mathrm{L}|_W + 1$$

Załóżmy że $|t_1|_L \leq |t_1|_W + 1$ oraz $|t_2|_L \leq |t_2|_W + 1$. Wówczas

$$|t_1 \ /\backslash\ t_2|_L = |t_1|_L + |t_2|_L \leq |t_1|_W + |t_2|_W + 2 = |t_1 \ /\backslash\ t_2|_W + 1$$

Na mocy twierdzenia o indukcji dla typu `tree` powyższa nierówność jest zatem prawdziwa dla dowolnego drzewa.

Pojedynczy krok indukcyjny polega na przejściu od dwóch drzew t_1 i t_2 do większego drzewa $t_1 \ /\backslash\ t_2$. Rekursja strukturalna jest również podporządkowana *strukturze* drzewa: wynik wywołania funkcji f dla drzewa $t_1 \ /\backslash\ t_2$ należy obliczyć na podstawie wyniku wywołania tej funkcji osobno dla drzew t_1 i t_2. Dla typu `tree` schemat rekursji strukturalnej wygląda zatem następująco:

$$\begin{aligned}&\texttt{fun}\ f\ \mathrm{L} = c\\ &\quad \mid\ f\ (t_1 \ /\backslash\ t_2) = g(t_1, t_2,\ f(t_1),\ f(t_2))\end{aligned}$$

Schemat iteracji otrzymamy odrzucając parametry t_1 i t_2 funkcji g:

$$\begin{aligned}&\texttt{fun}\ f\ \mathrm{L} = c\\ &\quad \mid\ f\ (t_1 \ /\backslash\ t_2) = g(f(t_1),\ f(t_2))\end{aligned}$$

Możemy zdefiniować uniwersalne funkcjonały — rekursor i iterator — realizujące rekursję i iterację strukturalną dla typu `tree`:

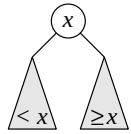
```
fun treeRec g c L = c
  | treeRec g c (t1 /\ t2) =
      g (t1, t2, treeRec g c t1, treeRec g c t2)
fun treeIter g c L = c
  | treeIter g c (t1 /\ t2) = g (treeIter g c t1, treeIter g c t2)
```

Wiele funkcji można łatwo zapisać z użyciem rekursji strukturalnej, np. funkcję `height` ujawniającą wysokość drzewa możemy zaprogramować następująco:

```
fun height L = 0
  | height (t1 /\ t2) = 1 + Int.max (height t1, height t2)
```

lub użyć iteratora:

```
val height = treeIter (fn (l,r) => 1 + Int.max(l,r)) 0
```



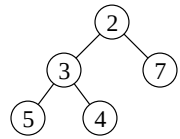
Rysunek 14.2. Binarne drzewo poszukiwań

14.2.3. Drzewa binarne o etykietowanych wierzchołkach

W tym przypadku mamy jeden liść i wiele wierzchołków wewnętrznych. Drzewo jest więc albo liściem, albo trójką złożoną z etykiety wierzchołka i dwóch poddrzew. Nie warto ustalać, czym są etykiety wierzchołków — wygodniej zdefiniować typ polimorficzny, którego parametrem będzie typ etykiet wierzchołków. Ponieważ mamy tylko jeden liść, gra on rolę „korka", za pomocą którego można zakończyć budowanie drzewa, podobnie jak `nil` dla list. Analogia z listami jest bardzo trafna. Jedyna różnica to ta, że lista niepusta jest parą złożoną z etykiety i *jednej* listy, tu zaś mamy *dwa* poddrzewa. Liść nazwiemy *drzewem pustym* i będziemy oznaczać `%`. Dla wierzchołków wewnętrznych przyjmiemy notację infiksową:

```
datatype 'a tree = $ of 'a * ('a tree * 'a tree) | %
infix 4 $
```

Np. drzewo



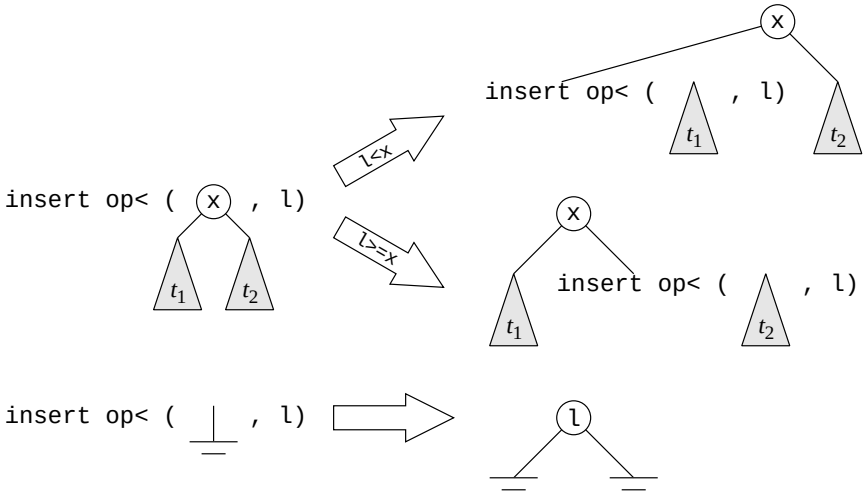
jest reprezentowane przez wartość

```
2$(3$(5$(%,%),4$(%,%)),7$(%,%))
```

typu `int tree`. Niekiedy jest wygodnie pomijać liście i traktować powyższe drzewa tak, jakby ich wierzchołki miały od zera do dwóch poddrzew.

14.2.4. Binarne drzewa poszukiwań

Jeżeli na wartościach typu σ określono relację porządku `op<` : σ `*` $\sigma \rightarrow$ `bool`, to `<` wraz z typem σ `tree` może posłużyć do budowy algorytmów przetwarzających *binarne drzewa poszukiwań*, tj. drzewa binarne etykietowane elementami typu σ w taki sposób, że etykiety wszystkich wierzchołków lewego poddrzewa są mniejsze (względem relacji `<`) od etykiety korzenia, a ta jest niewiększa od wszystkich etykiet prawego poddrzewa (por. rysunek 14.2) i własność ta zachodzi również dla obu jego poddrzew.



Rysunek 14.3. Wstawianie elementu do drzewa poszukiwań

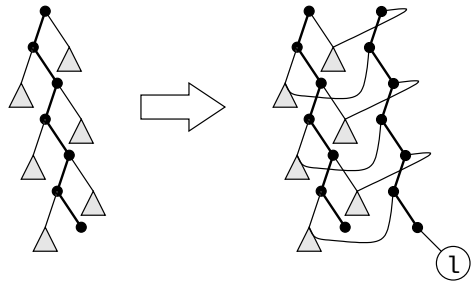
Operacje wstawiania i usuwania (z powtórzeniami) elementów z takich drzew wygodnie zaprogramować jako funkcje polimorficzne sparametryzowane relacją <. Wstawianie polega na zejściu w drzewie ścieżką aż do liścia zgodnie z relacją <, tak, by nowo wstawiony element nie zaburzył porządku. Wstawianie elementu do drzewa pustego (liścia) polega na utworzeniu korzenia, którego oba poddrzewa są puste (por. rysunek 14.3):

```
fun insert _ (l,%) = l$(%,%)
  | insert op< (l,x$(t1,t2)) =
      if l < x
        then x$(insert op< (l,t1), t2)
        else x$(t1, insert op< (l,t2))
```

Jeżeli drzewo jest w miarę dobrze wyważone, to ścieżka po której schodzi funkcja `insert` jest jedynie logarytmicznej długości względem rozmiaru drzewa. Taki jest więc również czas pracy funkcji `insert`. Pamięcią również gospodaruje ona oszczędnie. Nowa pamięć jest rezerwowana jedynie do wybudowania nowej ścieżki zakończonej wstawianym elementem (por. rysunek 14.4). Sytucja znacznie się pogarsza, jeżeli drzewo nie jest wyważone. W najgorszym przypadku, gdy drzewo jest pojedynczą nitką, `insert` działa w czasie liniowym i kopiuje całe drzewo (wówczas działa tak samo, jak wstawianie elementu na koniec listy).

Największy wierzchołek drzewa poszukiwań osiągniemy schodząc w dół drzewa po ścieżce wybierając prawe poddrzewa aż do napotkania wierzchołka nie posiadającego prawego poddrzewa (wynika to wprost z warunku na uporządkowanie drzewa, zobacz rysunek 14.5). Wybieranie największego elementu realizuje funkcja `maxel`:

```
exception Maxel
```



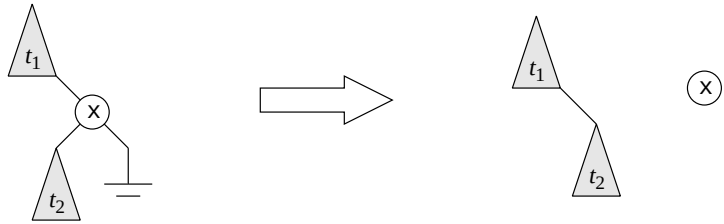
Rysunek 14.4. Funkcja `insert` zużywa pamięć jedynie do wybudowania nowej ścieżki zakończonej wstawianym elementem. Pozostałe poddrzewa są współdzielone ze starym drzewem

```
fun maxel (x$(t1,%)) = (x,t1)
  | maxel (x$(t1,t2)) =
      let
        val (y,t2') = maxel t2
      in
        (y,x$(t1,t2'))
      end
  | maxel % = raise Maxel
```

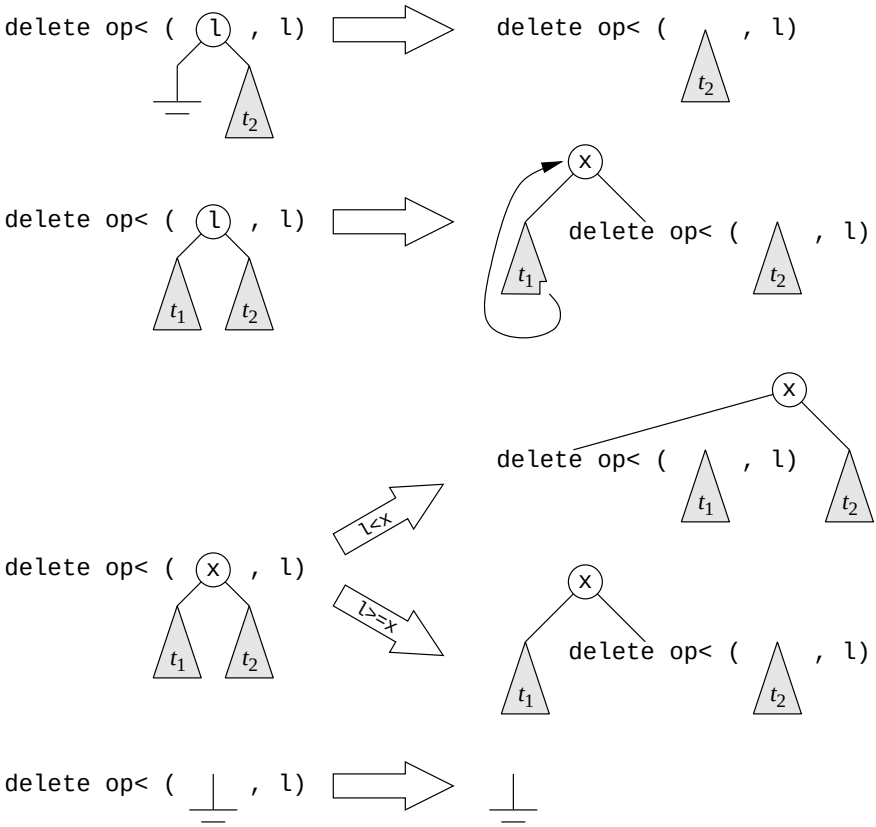
która zwraca parę złożoną z największej etykiety i drzewa z którego ją usunięto. Czas działania i zużycie pamięci są podobne jak dla funkcji `insert`. Ponieważ w drzewie pustym nie ma największego elementu, zadeklarowano wyjątek `Maxel`, zgłaszany w razie żądania wyznaczenia największego elementu drzewa pustego.

Usuwanie elementu z drzewa poszukiwań jest operacją ciut bardziej skomplikowaną (por. rysunek 14.6). Wszystkie usuwane wierzchołki o danej etykiecie (zakładamy że etykieta może się powtarzać — tak zdefiniowaliśmy funkcję `insert`) leżą na jednej ścieżce (jest to ta sama ścieżka, która zostałaby wybrana przez funkcję `insert` przy wstawianiu tej etykiety). Należy nią zejść aż do liścia usuwając po drodze znalezione wierzchołki. Jest to łatwe, jeśli usuwany wierzchołek nie ma lewego poddrzewa (wówczas po prostu usuwamy korzeń i kontynuujemy schodzenie w prawym poddrzewie). Jeżeli lewe poddrzewo usuwanego elementu jest niepuste, należy z pomocą funkcji `maxel` wybrać zeń największą etykietę i uczynić ją etykietą korzenia:

```
fun delete _ (_,%) = %
  | delete op< (l,x$(t1,t2)) =
      if l < x
        then x$(delete op< (l,t1), t2)
      else if x < l
        then x$(t1, delete op< (l,t2))
```



Rysunek 14.5. Wybranie największego elementu x polega na zejściu w drzewie t_1 na prawo aż do znalezienia wierzchołka nie posiadającego prawego poddrzewa. Usunięcie tego wierzchołka jest łatwe, ponieważ ma co najwyżej jedno niepuste poddrzewo



Rysunek 14.6. Usuwanie wszystkich wystąpień elementu z drzewa poszukiwań

```
      else (case t1 of
        % => delete op< (l,t2) |
        _ => let
               val (y,t1') = maxel t1
             in
               y$(t1', delete op< (l,t2))
             end)
```

Wyposażeni w możliwość wstawiania i usuwania elementów z drzewa możemy odbyć z komputerem np. poniższą rozmowę:

```
- val ins = insert op< : int * int tree -> int tree;
val ins = fn : int * int tree -> int tree
- val del  =  delete op< : int * int tree -> int tree;
val del = fn : int * int tree -> int tree
- ins(10,%);
val it = 10 $ (%,%) : int tree
- ins(3,it);
val it = 10 $ (3 $ (%,%),%) : int tree
- ins(20,it);
val it = 10 $ (3 $ (%,%),20 $ (%,%)) : int tree
- ins(3,it);
val it = 10 $ (3 $ (%,3 $ (%,%)),20 $ (%,%)) : int tree
- ins(20,it);
val it = 10 $ (3 $ (%,3 $ (%,%)),20 $ (%,20 $ (%,%))) : int tree
- ins(10,it);
val it = 10 $ (3 $ (%,3 $ (%,%)),20 $ (10 $ (%,%),20 $ (%,%)))
   : int tree
- del(3,it);
val it = 10 $ (%,20 $ (10 $ (%,%),20 $ (%,%))) : int tree
- del(20,it);
val it = 10 $ (%,10 $ (%,%)) : int tree
```

Ponieważ drzewo zawiera klucze w porządku niemalejącym, wystarczy je po zbudowaniu obejść infiksowo, by otrzymać ciąg posortowany. Zaprogramujemy do tego celu funkcję `flatten`, „spłaszczającą” drzewo do listy:

```
fun flatten (x$(t1,t2)) = flatten t1 @ [x] @ flatten t2
  | flatten % = nil
```

Co prawda rekursja w drzewie nie jest liniowa, a więc i nie jest ogonowa, jednak `flatten` warto zaprogramować z użyciem akumulatora i podwójnego wywołania rekurencyjnego w argumencie. Dzięki temu oszczędza się na pamięci marnowanej przez funkcję `@`:

```
fun flatten t =
  let
```

```
    fun aux acc (x$(t1,t2)) = aux (x :: aux acc t2) t1
      | aux acc % = acc
  in
    aux nil t
  end
```

Cały algorytm *Treesort* zmieści się już teraz w jednej linijce — z pomocą iteratora `foldr` budujemy drzewo wstawiając elementy listy `xs` zgodnie z relacją `op<`, a następnie obchodzimy drzewo infiksowo:

```
fun treesort op< = flatten o foldr (insert op<) %
```

Algorytm `treesort` jest stabilny, tj. nie zmienia kolejności równych elementów.

Spinanie list w funkcji `flatten` przypomina wyrażenie, które analizowaliśmy podczas programowania algorytmu *Quicksort* (opisanego na stronie 20). Istotnie, *Treesort* i *Quicksort* działają według tej samej zasady, różnią się jedynie użytymi strukturami danych. Podczas obliczania funkcji `qsort` również jest budowane drzewo i to dokładnie takiej samej postaci jak jawnie budowane drzewo w funkcji `treesort` — jego wierzchołkami są rekordy aktywacji kolejnych wywołań rekurencyjnych.

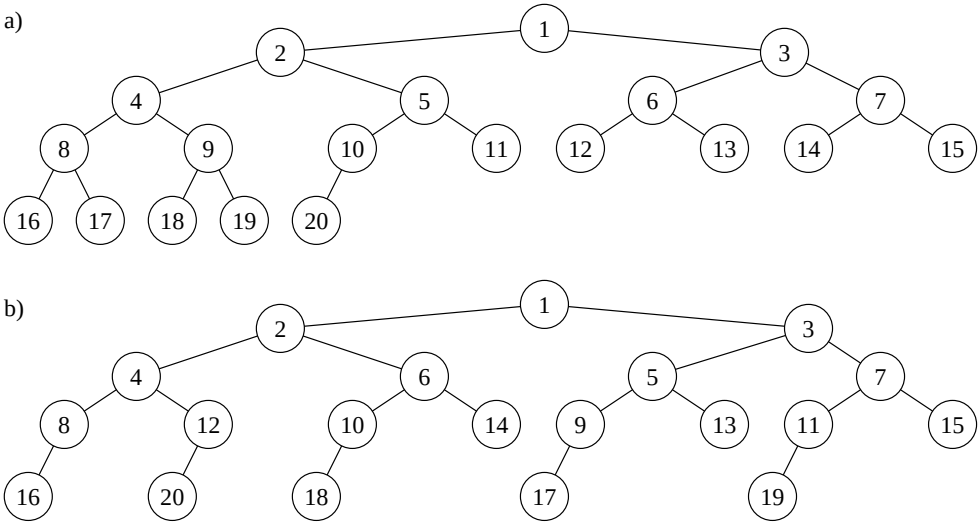
14.2.5. Sterty

Definicja 14.2. Drzewo $\langle w, \langle t_1, \ldots, t_n\rangle\rangle$ jest *wyważone*, jeśli $|h(t_i) - h(t_j)| \leq 1$ dla $i, j = 1, \ldots, n$, gdzie $h(t)$ jest wysokością drzewa t i jeśli drzewa $t_1, \ldots, t_n$ są wyważone. Liść l jest drzewem wyważonym.

Złośliwe dane (np. ciąg posortowany na wejściu algorytmu `treesort`) powodują, że budowane drzewo nie jest wyważone i w najgorszym przypadku może przybrać postać listy. Wówczas efektywność algorytmów znacznie się pogarsza. Aby zadbać o wyważenie binarnych drzew poszukiwań wymyślono nietrywialne algorytmy (np. AVL). W wielu przypadkach jednak nie potrzeba aż tak uporządkowanych drzew — wystarczą *sterty*. Drzewo $\langle w, \langle t_1, \ldots, t_n\rangle\rangle$, o wierzchołkach wewnętrznych etykietowanych elementami typu σ uporządkowanego relacją $\texttt{op}\leq : \sigma * \sigma \to \texttt{bool}$ jest *stertą* (albo *stogiem*), jeśli jest drzewem wyważonym oraz $w_i \leq w$, $i = 1, \ldots, n$, gdzie w_i jest korzeniem t_i (pomijamy te t_i, które są liśćmi). Drzewo puste jest stertą. Zwykle będziemy zakładać, że sterta jest drzewem binarnym.

Wstawianie elementu do sterty jest dużo prostsze niż do drzewa poszukiwań. Aby zapewnić wyważenie, numerujemy wierzchołki drzewa wierszami (zob. rysunek 14.7a) i żądamy, by drzewo o n wierzchołkach zawierało wierzchołki o numerach $1 \div n$. Miejsce na wstawienie elementu do n-elementowej sterty jest zatem jednoznacznie określone. Aby zachować warunek sterty, należy jeszcze wymienić miejscami etykiety wierzchołków leżących na ścieżce od wstawionego elementu do korzenia. Razem z drzewem przechowujemy zatem jeszcze liczbę elementów sterty:

```
datatype 'a heap = Heap of int * 'a tree
val empty = Heap (0,%)
```



Rysunek 14.7. Dwudziestoelementowa sterta z elementami etykietowanymi wierszami zgodnie z a) naturalną i b) odwróconą kolejnością bitów

Ścieżka od korzenia do *n*-tego elementu jest zadana przez binarne rozwinięcie liczby *n* po odrzuceniu najstarszego bitu, np. 19 element sterty osiągniemy wybierając dwukrotnie lewe (00) a następnie dwukrotnie prawe (11) poddrzewo korzenia, ponieważ rozwinięciem binarnym liczby 19 jest 1 0011 (rys. 14.7a). Ponieważ dużo łatwiej obliczać kolejne bity liczby poczynając od bitu najmniej znaczącego (używając operacji `mod` i `div`), to stertę zwykle numeruje się w takiej kolejności (zob. rysunek 14.7b). Ostatecznie funkcja wstawiająca element do sterty ma postać:

```
exception Impossible of string
fun impossible message = raise Impossible message
exception WrongHeap
fun insert op< (l, Heap (n,x$(t1,t2))) =
      if n mod 2 = 0
        then (case insert op< (l, Heap ((n-1) div 2, t2)) of
          Heap (_,y$(tl,tr)) =>
            if x < y
              then Heap (n+1, y$(t1,x$(tl,tr)))
              else Heap (n+1, x$(t1,y$(tl,tr))) |
          _ => impossible "insert")
        else (case insert op< (l, Heap (n div 2, t1)) of
          Heap (_,y$(tl,tr)) =>
            if x < y
```

```
              then Heap (n+1, y$(x$(tl,tr),t2))
              else Heap (n+1, x$(y$(tl,tr),t2)) |
          _ => impossible "insert")
  | insert _ (l, Heap (0,%)) = Heap (1, l$(%,%))
  | insert _ _ = raise WrongHeap
```

Ponieważ w wyrażeniu `case` analizujemy stertę, do której właśnie wstawiliśmy element, jest ona na pewno niepusta. Jednak kompilator nie jest tego w stanie udowodnić (ani my nie jesteśmy w stanie tego udowodnić kompilatorowi) i ostrzega, że jeden wzorzec nie wystarczy. Pomimo iż jesteśmy pewni, że o ile program jest poprawny, drugi wariant nigdy się nie zdarzy, wygodnie zdefiniować specjalną funkcję `impossible` wywoływaną w razie gdyby zdarzyła się taka nieprzewidziana sytuacja. Program może być błędny!

Nie wszystkie dane typu `'a Heap` są poprawnymi stertami. Wyjątek `WrongHeap` jest zgłaszany przy próbie wstawienia elementu do takiej danej.

Zupełnie podobnie programujemy funkcję `maxel` usuwającą największy element sterty. Algorytm *Heapsort* buduje wpierw stertę, a następnie usuwa z niej cyklicznie największy element aż do jej opróżnienia:

```
fun heapsort op< =
  let
    fun takeIt acc (Heap(0,%)) = acc
      | takeIt acc h =
          let
            val (m,h') = maxel h
          in
            takeIt (m::acc) h'
          end
  in
    takeIt nil o foldr (insert op<) empty
  end
```

Pesymistyczny czas pracy funkcji `heapsort` dla listy długości n, w odróżnieniu od funkcji `qsort`, jest rzędu $n \log n$ (dla `qsort` jest rzędu n^2).

Sterty można wykorzystać również do implementowania tablic o „bezpośrednim" dostępie. Mówi się, że czas dostępu do elementu tablicy w języku imperatywnym jest stały, niezależny od wielkości tablicy. Takie rozumowanie *implicite* zakłada, że maszyna wykonuje operacje arytmetyczne (dodanie *offsetu* do początku tablicy) na adresach w stałym czasie, a więc, że rozmiar komórek pamięci jest nieograniczony. Przy przyjęciu bardziej realistycznego logarytmicznego kryterium kosztów, czas adresowania pośredniego wynosi $O(\log n)$, gdzie n jest rozmiarem tablicy. Pamiętając elementy w wyważonym drzewie zgodnie z rysunku 14.7 możemy zaprogramować strukturę danych przypominającą tablicę o koszcie wstawienia, zmiany, usunięcia i ujawnienia elementu wynoszącym $O(\log n)$, więc asymptotycznie tak samo dobrym, jak dla imperatywnej tablicy przy logarytmicznym kryterium kosztów. Szczegóły można znaleźć np. w książce Paulsona [135].

Najtrudniej zaimplementować struktury danych, które jednocześnie mogą ulegać zmianie (ang. *mutable*) i być trwałe (ang. *persistent*). Tablice tego rodzaju implementuje się w postaci stert. Typy implementujące tablice i spełniające tylko jeden z powyższych warunków są dostępne w Bibliotece Standardowej SML-a w strukturach `Vector` i `Array`. Są tam zdefiniowane abstrakcyjne typy danych `'a vector` i `'a array` i zestaw funkcji do przetwarzania wartości tych typów. Wektory są trwałe, ale nie można ich modyfikować, zaś tablice można modyfikować, ale są ulotne. Wektory nadają się do reprezentowania stałych danych, które łatwo ponumerować i z których często się korzysta. Tablice zaś odpowiadają tablicom znanym z imperatywnych języków programowania. Do zapisywania wektorów i tablic przewidziano w SML-u specjalne konstrukcje składniowe podobne do zapisu list:

$$\texttt{\#[}x_1, \ldots, x_n\texttt{]}$$
$$\texttt{[|}x_1, \ldots, x_n\texttt{|]}$$

Pierwsza z nich oznacza wektor, druga zaś tablicę elementów $x_1, \ldots, x_n$. Napisy `[|`, `|]` i `#[` są słowami kluczowymi, oba znaki nie mogą być więc rozdzielone spacją.

14.3. Abstrakcyjne drzewa rozbioru w SML-u

Deklaracja `datatype` pozwala reprezentować dowolnie skomplikowane dane, byleby miały strukturę drzewa, w szczególności wyrażenia. Rozpatrzmy dla przykładu wyrażenia arytmetyczne złożone z liczb całkowitych, zmiennych, podstawowych działań arytmetycznych i funkcji trygonometrycznych:

```
infix 6 ++
infix 7 **
datatype expr = Real of real | Var of string |
                ++ of expr * expr | ** of expr * expr |
                Sin of expr | Cos of expr
```

(typ danych został uproszczony, by łatwiej analizować program). Za zmienną można w wyrażeniu podstawić inne wyrażenie:

```
infix 9 subst
fun (e1 as Var x') subst (x,e) = if x'=x then e else e1
  | (e1 ++ e2) subst p = e1 subst p ++ e2 subst p
  | (e1 ** e2) subst p = e1 subst p ** e2 subst p
  | (Sin e1) subst p = Sin (e1 subst p)
  | (Cos e1) subst p = Cos (e1 subst p)
  | e1 subst _ = e1
```

Dla przykładu wyrażenie

```
Real 2.0 ** Sin (Var "x") ** Cos (Var "x")
```

typu `expr` reprezentuje wyrażenie $2 \sin x \cos x$. Jeżeli za x podstawimy y^2, to otrzymamy wyrażenie $2 \sin(y^2) \cos(y^2)$:

```
- (Real 2.0 ** Sin (Var "x") ** Cos (Var "x")) subst
= ("x", Var "y" ** Var "y");
val it = Real 2.0 ** Sin (Var "y" ** Var "y") **
  Cos (Var "y" ** Var "y") : expr
```

O ile wyrażenie nie zawiera zmiennych, jego wartość daje się obliczyć:

```
fun eval (e1 ++ e2) =
    (case (eval e1, eval e2) of
      (Real r1, Real r2) => Real (r1+r2) |
      (e1,e2) => e1 ++ e2)
  | eval (e1 ** e2) =
    (case (eval e1, eval e2) of
      (Real r1, Real r2) => Real (r1*r2) |
      (e1,e2) => e1 ** e2)
  | eval (Sin e) =
    (case eval e of
      Real x => (Real o Math.sin) x  |
      e => e)
  | eval (Cos e) =
    (case eval e of
      Real x => (Real o Math.cos) x |
      e => e)
  | eval e = e
```

W szczególności możemy obliczyć wartość funkcji, gdy jej parametr ma zadaną wartość:

```
infix 8 at
fun f at x = eval (f subst x)
```

Korzystając ze wzorów na pochodne funkcji elementarnych, możemy napisać funkcję `d`, obliczającą pochodną wyrażenia względem zadanej zmiennej:

```
infix 8 d
fun (Real _) d _ = Real 0.0
  | (Var x') d x = if x'=x then Real 1.0 else Real 0.0
  | (e1 ++ e2) d x = e1 d x ++ e2 d x
  | (e1 ** e2) d x = e1 d x ** e2 ++ e1 ** e2 d x
  | (Sin e) d x = Cos e ** e d x
  | (Cos e) d x = Real (~1.0) ** Sin e ** e d x
```

Umiejąc obliczać pochodne, możemy rozwinąć funkcję w szereg Maclaurina do n-tego wyrazu:

$$f(x) \quad \approx \quad \sum_{k=0}^{n} \frac{f^{(k)}(0)}{k!} x^k$$

gdzie $f^{(k)}$ jest k-tą pochodną f względem zmiennej x. W SML-u zaprogramujemy funkcję `maclaurin` otrzymującą jako argumenty parametr n, zmienną x i wyrażenie opisujące funkcję f:

```
fun maclaurin n x f =
  let
    fun aux (acc, fack, xk, edxk, k) =
      if k > n
        then acc
        else
        let
          val coeff = eval (Real (1.0/fack) **
                      (edxk subst (x, Real 0.0)))
        in
          aux (acc ++ coeff ** xk,
               fack * real (k+1),
               (Var x) ** xk,
               edxk d x,
               k+1)
        end
  in
    aux (Real 0.0, 1.0, Real 1.0, f, 0)
  end
```

Rozwińmy dla przykładu w szereg funkcję $2 \sin x \cos x$:

```
- val f = Real 2.0 ** Sin (Var "x") ** Cos (Var "x");
val f = Real 2.0 ** Sin (Var "x") ** Cos (Var "x") : exp
- val f' = eval (maclaurin 5 "x" f);
val f' =
  Real 0.0 ++ Real 2.0 ** (Var "x" ** Real 1.0) ++
  Real 0.0 ** (Var "x" ** (Var "x" ** Real 1.0)) ++
  Real ~1.33333333333333 **
     (Var "x" ** (Var "x" ** (Var "x" ** Real 1.0))) ++
  Real 0.0 ** (Var "x" ** (Var "x" **
        (Var "x" ** (Var "x" ** Real 1.0)))) ++
  Real 0.266666666666667 **
     (Var "x" ** (Var "x" ** (Var "x" **
        (Var "x" ** (Var "x" ** Real 1.0))))) : expr
```

Możemy wyliczyć wartość funkcji `f'` dla `x` np. równego `0.3`:

```
- f' at ("x", Real 0.3);
val it = Real 0.564648 : expr
```

Przybliżenie jest nie najgorsze:

```
- Math.sin 0.6;
val it = 0.564642473395035 : real
```

Niestety problem upraszczania wyrażeń zawierających funkcje przestępne jest nierozstrzygalny (dokładniej nie sposób napisać algorytmu odpowiadającego na pytania typu: „czy $\sin(2x)$ jest równe $2 \sin x \cos x$?”). Można próbować stosować różne heurystyki.

Przydałoby się wyposażyć powyższy program w prosty parser wyrażeń i uruchomić interakcyjną pętle, tak, by użytkownik mógł wprowadzać wyrażenia i żądać symbolicznego obliczenia pochodnych, całek (!) itp.

14.4. Zadania

Zadanie 14.1. Zdefiniuj typy książek i czytelników dla problemu opisanego na stronie 244 nie odwołując się do rekursji wzajemnej. Czy Twoja definicja jest lepsza, czytelniejsza, prostsza?

Zadanie 14.2. Napisz konkretną gramatykę wyrażeń dla gramatyki abstrakcyjnej przedstawiobej na stronie 244 i parser tych wyrażeń, tj. funkcję typu `string -> wyrazenie`.

Zadanie 14.3. Zdefiniuj wartości typów `mezczyzna` i `kobieta` opisanych na stronie 245 będące nazwami bohaterów Ks. Rodzaju 4:1–26 i 5:1–32.

Zadanie 14.4. Dla typów z poprzedniego zadania zaprogramuj funkcje

```
sa_bracmi      : mezczyna * mezczyzna -> bool
sa_rodzenstwem : mezczyzna * kobieta -> bool
jest_ciotka    : kobieta * mezczyzna -> bool
jest_babcia    : kobieta * mezczyzna -> bool
jest_kuzynka   : kobieta * mezczyzna -> bool
```

Rozdział 15

Moduły

Zapisanie programu w jednym pliku w postaci ciągu deklaracji jest możliwe tylko w przypadku prostych zagadnień. Większe kolekcje procedur trzeba gromadzić w postaci bibliotek. Biblioteka (moduł, pakiet, struktura — nazwy bywają różne w zależności od języka) powinna składać się z części publicznej, opisującej zawartość biblioteki i części prywatnej, implementującej odpowiednie procedury. Zawartość części prywatnej powinna być niedostępna dla użytkownika modułu, tak, by nie wykorzystywał on konkretnych cech danej implementacji a używał jedynie (abstrakcyjnych) własności opisanych w części publicznej. Obie części powinny być od siebie w miarę niezależne, w szczególności nie powinny być powiązane składniowo w jednym pliku.

Do uzupełnienia:

15.1. Oddzielenie specyfikacji i implementacji: Modula 2
15.2. Rodzajowość w Adzie

15.3. Moduły w SML-u

System modułów SML-a jest powszechnie uważany za najdoskonalszy, najbardziej rozbudowany i wyrafinowany system modularyzacji programów, jaki kiedykolwiek zaimplementowano w rzeczywistym języku programowania. Definicja języka SML traktuje system modułów jako osobną, wydzieloną część języka. Część SML-a nie zawierającą modułów nazywa się Core ML-em. Omówiliśmy ją w poprzednich rozdziałach.

15.3.1. Motywacja

Wykorzystując jedynie Core ML opiszemy po krótce na czym polega modularyzacja programu i dlaczego Core ML nie wystarcza do jej pełnej realizacji.

Rozważmy zestaw prostych funkcji arytmetycznych: `succ` obliczającą następnik liczby typu `int`, `pred` obliczającą poprzednik i `double` podwajającą swój argument. Odpowiednie

deklaracje możemy umieścić w osobnym pliku, wczytywanym do środowiska za pomocą funkcji `use`. Jesteśmy w stanie zapanować nad taką „biblioteką”, jeśli zawiera dziesięć lub dwadzieścia deklaracji. W przypadku setek funkcji powstałby zupełny chaos. Dlatego funkcje powiązane ze sobą w jakiś sposób dobrze byłoby zgromadzić w postaci osobno nazwanej kolekcji wartości. W SML-u mogłyby do tego posłużyć rekordy:

```
- type INTLIB =
=   {
=     succ   : int -> int,
=     pred   : int -> int,
=     double : int -> int
=   }
= val IntLib : INTLIB =
=   {
=     succ   = fn x => x+1,
=     pred   = fn x => x-1,
=     double = fn x => 2*x
=   };
type  INTLIB = {double:int -> int, pred:int -> int, succ:int -> int}
val IntLib = {double=fn,pred=fn,succ=fn} : INTLIB
```

Typ `INTLIB` jest niejako częścią publiczną naszego „modułu”, rekord `IntLib` — częścią prywatną. Użytkownikowi wystarczy znajomość typu `INTLIB`, by mógł z naszej biblioteki korzystać:

```
- val user = {mkOdd = fn n => ((#succ IntLib) o (#double IntLib)) n};
val user = {mkOdd=fn} : {mkOdd:int -> int}
```

Teraz zamiast anonimowych funkcji `succ` i `double` piszemy

```
#succ intLib i #double intLib
```

jawnie wskazując, że są elementami „biblioteki” `IntLib` i dlatego łatwo jest je odszukać w programie. Nadto system sprawdza, czy implementator biblioteki wywiązał się z zadania, tj. czy zaprogramował wszystkie funkcje wymienione w części publicznej i czy mają one zadeklarowane wcześniej typy. Implementację można zmieniać wielokrotnie i dopóki dopasowuje się do typu `INTLIB`, wystarczy skompilować program jeszcze raz, żadne zmiany w programie użytkownika nie są konieczne.

Korzystając z definicji typu `INTLIB` użytkownik może programować i *kompilować* swój program zanim jeszcze implementacja rekordu `IntLib` zostanie dostarczona (np. przez innego programistę). Wystarczy tę implementację uczynić *parametrem* programu użytkownika:

```
- fun generic_user (IntLib : INTLIB) =
=     {mkOdd = fn n => ((#succ IntLib) o (#double IntLib)) n};
val generic_user = fn : INTLIB -> {mkOdd:int -> int}
```

Program `generic_user` dało się skompilować, mimo iż biblioteka `IntLib` jeszcze nie istnieje! Specyfikacja została oddzielona od implementacji. Z chwilą dostarczenia biblioteki `IntLib` programu nie trzeba kompilować powtórnie, wystarczy utworzyć jego *instancję* aplikując program *rodzajowy* do parametru:

```
- val user = generic_user (IntLib);
val user = {mkOdd=fn} : {mkOdd:int -> int}
```

Nie zachęcam nikogo do modularyzowania swoich programów przy użyciu rekordów i funkcji. Chodziło mi o pokazanie pewnej analogii. Odpowiedniość pomiędzy elementarnymi konstrukcjami języka i mechanizmami tworzenia modułów jest następująca:

typy = części publiczne (specyfikacje) modułów,

rekordy = części prywatne (implementacje) modułów,

funkcje = moduły rodzajowe (moduły sparametryzowane innymi modułami), tj. funkcje przeprowadzające implementacje jednych modułów w implementacje innych modułów.

15.3.2. Struktury, sygnatury i funktory

Do programowania części prywatnych modułów używa się w SML-u *struktur*, niezwykle podobnych do rekordów. Zamiast nawiasów klamrowych pisze się słowa kluczowe `struct` i `end`, pomiędzy którymi zamiast pól rekordu wypisuje się zwykłe deklaracje `val` (oraz wszelkie inne deklaracje, o czym dalej):

```
struct
  val succ = fn n => n+1
  val pred = fn n => n-1
  fun double n = 2*n
end
```

Jak widać zamiast deklaracji `val` można też używać deklaracji `fun`. Powyższa struktura żyje w zupełnie innym świecie niż poznane wcześniej wartości, nie może więc być związana z nazwą za pomocą deklaracji `val`. Jest natomiast odpowiednik deklaracji `val` w postaci deklaracji `structure`:

```
- structure IntLib =
=   struct
=     fun succ   x = x+1
=     fun pred   x = x-1
=     fun double x = 2*x
=   end;
structure IntLib :
  sig
    val double : int -> int
```

```
    val pred : int -> int
    val succ : int -> int
  end
```

Do wartości zdefiniowanych w strukturze odwołujemy się za pomocą prefiksowania ich nazw nazwą struktury zakończoną kropką, np. `IntLib.succ`. Nazwy obiektów prefiksowane nazwami struktur to tzw. *długie identyfikatory* (ang. *long identifiers*). Dla zagnieżdżonych struktur można napisać całą ścieżkę dostępu do wartości, np.

```
- Posix.FileSys.S.irusr;
val it = MODE 0wx100 : ?.POSIX_FileSys.S.mode
```

Teraz możemy napisać program użytkownika, również w postaci struktury:

```
- structure user =
=   struct
=     fun mkOdd n = (IntLib.succ o IntLib.double) n
=   end;
structure user : sig val mkOdd : int -> int end
```

Potwierdzając deklarację struktury system wypisuje coś na kształt jej typu, tzw. *sygnaturę* struktury. Podobnie jak typy wartości, najogólniejsze sygnatury są automatycznie rekonstruowane przez system. Ich postać jest podobna do typu rekordowego: zamiast nawiasów klamrowych pisze się słowa `sig` i `end`, pomiędzy którymi znajduje się lista *specyfikacji*. *Specyfikacje wartości* mają postać:

$$\texttt{val}\ x : \sigma$$

gdzie x jest identyfikatorem wartości a σ — jej typem. Sygnatura przypomina wyrażenie typowe, ale żyje w innej przestrzeni, dlatego nie można jej związać z nazwą za pomocą deklaracji `type`. Mamy natomiast odpowiednik deklaracji `type` w postaci deklaracji `signature`:

```
- signature INTLIB =
=   sig
=     val pred   : int -> int
=     val succ   : int -> int
=     val double : int -> int
=   end;
signature INTLIB =
  sig
    val pred : int -> int
    val succ : int -> int
    val double : int -> int
  end
```

Deklarując strukturę możemy jawnie wymusić sprawdzenie, czy struktura *dopasowuje się* do danej sygnatury, podobnie jak to czyniliśmy dla typów:

```
- structure IntLib : INTLIB =
=   struct
=     fun double x = 2*x
=     fun pred   x = x-1
=     fun succ   x = x+1
=   end;
structure IntLib : INTLIB
```

Typ wartości polimorficznej możemy *zawęzić* poprzez jawne wymuszenie typu:

```
- fun id x : int = x;
val id = fn : int -> int
- id true;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: int
  operand:         bool
  in expression:
    id true
```

Funkcja `id` otrzymuje typ `int` $\rightarrow$ `int`, mimo iż mogłaby mieć (ogólniejszy) typ $\alpha \rightarrow \alpha$. Nie można już jej użyć do argumentu typu `bool`. Podobnie działa tzw. *zawężenie sygnatury* (ang. *signature constraint*):

```
- signature INT =
=   sig
=     val id : int -> int
=     val zero : int
=   end
= structure Int : INT =
=   struct
=     fun id x = x
=     val zero = 0
=     fun succ x = x+1
=   end;
signature INT =
  sig
    val id : int -> int
    val zero : int
  end
structure Int : INT
```

Najogólniejszą sygnaturą dla struktury

```
struct
  fun id x = x
  val zero = 0
```

```
  fun succ x = x+1
end
```

jest sygnatura

```
sig
  val id : 'a -> 'a
  val zero : int
  val succ = int -> int
end
```

Na skutek zawężenia sygnatury za pomocą sygnatury `INT`, wartość `Int.id` otrzymuje typ `int` $\to$ `int` i (uwaga: nowość!) wartość `Int.succ` przestaje być widoczna:

```
- Int.id;
val it = fn : int -> int
- Int.succ;
Error: unbound variable or constructor: succ in path Int.succ
```

Tu analogia z rekordami przestaje działać: typu rekordu nie możemy zawęzić przez pominięcie któregoś z jego pól (jest to dosyć ciekawe i teoretycznie możliwe, wywołałoby jednak spore komplikacje implementacyjne).

Struktura S dopasowuje się zatem do sygnatury Σ jeśli zawiera (co najmniej) wszystkie wartości wyspecyfikowane w sygnaturze Σ i jeśli ich typy dopasowują się do typów podanych w sygnaturze Σ.

Zawężenia sygnatury używamy do ukrywania przed użytkownikiem detali implementacyjnych. Nawet jeśli chcemy pozostawić najogólniejszą sygnaturę, warto ją osobno zadeklarować, ponieważ pełni rolę części publicznej, specyfikacji naszego modułu.

Kontynuując analogię z rekordami z poprzedniego paragrafu, oczekujemy, że dla struktur również istnieją specjalnego rodzaju „funkcje". Istotnie, odwzorowania przeprowadzające struktury w inne struktury nazywamy *funktorami* i definiujemy w podobny sposób jak funkcje. Funktor nie może być zdefiniowany anonimowo (nie ma odpowiednika wyrażenia `fn`). Odpowiednikiem deklaracji `fun` jest deklaracja `functor` postaci

$$\texttt{functor}\ f\ (s : \Sigma_s) : \Sigma_T\ \texttt{=}\ T$$

gdzie f jest nazwą deklarowanego właśnie funktora, s jest identyfikatorem struktury będącej jego parametrem, T jest strukturą będącą wynikiem aplikacji funktora do argumentu, a Σ_s i Σ_T to sygnatury (identyfikatory lub wyrażenia postaci `sig` ... `end`) odpowiednio struktury s i wyniku T (zatem struktura T musi dopasowywać się do sygnatury Σ_T). W treści struktury T można się odwoływać do struktury s. Zauważmy, jak bardzo deklaracja funktora przypomina deklarację funkcji postaci

$$\texttt{fun}\ f\ (x : \sigma) : \tau\ \texttt{=}\ M$$

Sygnaturę Σ_T wraz z poprzedzającym ją dwukropkiem można pominąć. Wówczas system zrekonstruuje dla wyniku funktora najogólniejszą sygnaturę. Sygnatury parametru pominąć

nie wolno. Przykład rodzajowego programu użytkownika z poprzedniego paragrafu można przy pomocy funktora zapisać następująco:

```
- functor generic_user (IntLib : INTLIB) =
=   struct
=     fun mkOdd n = (IntLib.succ o IntLib.double) n
=   end;
functor generic_user : <sig>
```

W definicji funktora możemy nie odwoływać się do sygnatury `INTLIB` (części publicznej naszej biblioteki), jawnie specyfikując, z czego korzystamy:

```
- functor generic_user
=   (IntLib : sig
=               val succ : int -> int
=               val double : int -> int
=             end) =
=   struct
=     fun mkOdd n = (IntLib.succ o IntLib.double) n
=   end;
functor generic_user : <sig>
```

Obecnie argument funktora dopasuje się do każdej struktury w której zdefiniowano (co najmniej) dwie wartości `succ` i `double`, obie (co najmniej) typu `int` $\rightarrow$ `int`. Takie rozwiązanie bywa niekiedy wygodne, lepiej jednak jawnie używać zdefiniowanych wcześniej sygnatur.

Utworzenie gotowej struktury `user` jest już dziecinnie łatwe (pod warunkiem, że mamy już zdefiniowaną strukturę `IntLib`):

```
- structure user = generic_user (IntLib);
structure user : sig val mkOdd : int -> int end
```

Struktury mogą być elementami innych struktur (podobnie jak rekordy mogą być elementami innych rekordów). Odpowiednia specyfikacja struktury w sygnaturze ma postać:

$$\texttt{structure}\ s : \Sigma$$

gdzie s jest nazwą struktury, a Σ — jej sygnaturą. W celu ograniczenia zasięgu obowiązywania deklaracji struktury, można umieścić ją wewnątrz innej struktury i przesłonić za pomocą zawężenia sygnatury lub użyć deklaracji `local`.

Deklaracja sygnatury może wystąpić jedynie na zewnętrznym poziomie programu, nie może być uwikłana w konteksty lokalne i nie może być częścią struktury. W kompilatorze SML/NJ dopuszczono, by deklaracja sygnatury występowała pomiędzy słowami `local` oraz `in` w deklaracji `local`. Standard języka przewiduje, że również funktory muszą być deklarowane globalnie. W implementacji SML/NJ wprowadzono możliwość lokalnych deklaracji funktorów. Spowodowało to znaczne rozszerzenie języka, którego jednak nie będziemy w tym miejscu opisywać.

Deklaracji sygnatur, struktur i funktorów nie można mieszać z wartościami. Moduły to jakby wyższy poziom: wartości mogą być częściami składowymi struktur, jednak struktury *nie* mogą być częściami składowymi wartości. Dlatego żadne z powyższych trzech deklaracji nie mogą wystąpić w wyrażeniu `let` dostarczającym wartości. Mamy natomiast analogiczne wyrażenie `let`, którego wynikiem jest struktura. W tym wyrażeniu można używać deklaracji struktur (a w SML/NJ także sygnatur i funktorów), np.

```
signature TSIG = sig val y : int end
structure S =
  let
    val x = 5
    structure T : TSIG = struct val y = x end
  in
    T
  end
```

Deklaracje `structure`, `signature` i `functor` można, tak jak wszystkie inne deklaracje, łączyć słowem kluczowym `and`. Identyfikatory struktur, sygnatur i funktorów zostają udostępnione w środowisku dopiero po zakończeniu opracowania ich deklaracji, dlatego nie ma możliwości użycia definicji rekurencyjnych.

Podsumowując analogię z rekordami należy wskazać następujące istotne różnice:

- Funktory przypominają funkcje, ale przeprowadzają jedynie struktury w struktury[1] i nie mogą być definiowane rekurencyjnie.
- Zawężenie sygnatury struktury może polegać na zasłonięciu niektórych pól, co jest niemożliwe przy zawężaniu typu rekordu.
- Elementami struktury mogą być nie tylko wartości (jak w rekordach), ale także typy (por. następne paragrafy).

Uwagi. Nazwy *sygnatura* i *struktura* pochodzą z algebry (por. rodział 8). Istotnie, na część publiczną modułu można patrzeć jak na sygnaturę, a na odpowiadającą jej część prywatną jak na algebrę (strukturę algebraiczną) o danej sygnaturze. Odwzorowania przekształcające jedne algebry w inne nazywa się w algebrze (a przede wszystkim w teorii kategorii) *funktorami*.

15.3.3. Dodatkowa składnia parametru funktora

Jeżeli zechcielibyśmy przekazać funktorowi f jako parametr więcej niż jedną strukturę, np. dwie struktury $s_1 : \Sigma_1$ i $s_2 : \Sigma_2$, to możemy *ad hoc* wyspecyfikować nową strukturę s stanowiącą „opakowanie” dla obu struktur s_1 i s_2 i będącą parametrem funktora f:

$$\texttt{functor } f \texttt{ (}s \texttt{ : sig structure } s_1 \texttt{ : } \Sigma_1 \texttt{ and } s_2 \texttt{ : } \Sigma_2 \texttt{ end) = } S$$

a następnie aplikować go do struktury zbudowanej *ad hoc* z jego parametrów, np.

$$\texttt{structure } S' \texttt{ = } f \texttt{ (struct structure } s_1 \texttt{ = } S_1 \texttt{ and } s_2 \texttt{ = } S_2 \texttt{ end)}$$

[1] Dodatkowa, opisana dalej składnia może być pod tym względem myląca. Ponadto w SML/NJ parametrem funktora może być nie tylko struktura, ale także funktor.

W ciele funktora f do struktur s_1 i s_2 należy się odwoływać prefiksując ich nazwy nazwą struktury s, np. $s.s_1$ itp. Nie jest to zbyt wygodne. Dlatego wprowadzono dodatkową składnię, polegającą, z grubsza mówiąc, na uproszczeniu powyższej techniki zapisu wielu parametrów. W nawiasach ograniczających parametr deklaracji funktora może znaleźć się lista specyfikacji tak, jakby była otoczona słowami `sig` i `end`. W jego aplikacji musi wówczas wystąpić lista odpowiednich deklaracji tak, jakby były ograniczone słowami `struct` i `end`. Powyższy przykład w dodatkowej składni zapiszemy zatem jako:

$$\texttt{functor } f \texttt{ (structure } s_1 \texttt{ : } \Sigma_1 \texttt{ and } s_2 \texttt{ : } \Sigma_2\texttt{) = } S$$
$$\texttt{structure } S' \texttt{ = } f \texttt{ (structure } s_1 \texttt{ = } S_1 \texttt{ and } s_2 \texttt{ = } S_2\texttt{)}$$

Wszystkie specyfikacje wymienione w definicji funktora są widoczne w jego ciele (w tym przypadku są widoczne obie struktury s_1 i s_2).

Obu rodzajów składni nie wolno mieszać. Jeżeli zdefiniujemy funktor następująco:

```
signature SIG = sig val x : int end
structure Str : SIG = struct val x = 13 end
functor F (S : SIG) = struct fun f _ = S.x end
```

wówczas można go zaaplikować tylko w „stary” sposób:

```
- structure S' = F (Str);
structure S' : sig val f : 'a -> int end
- structure S' = F (structure S = Str);
Error: unmatched value specification: x
```

„Nowy” sposób jest niedopuszczalny. I odwrotnie, definiując funktor w „nowy” sposób:

```
functor F (structure S : SIG) = struct fun f _ = S.x end
```

możemy go używać jedynie w „nowy” sposób:

```
- structure S' = F (structure S = Str);
structure S' : sig val f : 'a -> int end
- structure S' = F (Str);
Error: unmatched structure specification: S
```

Dodatkowej składni używa się zwykle wówczas, gdy parametrem funktora jest więcej niż jedna struktura lub gdy chcemy funktorowi przekazać jako parametr nie strukturę a np. wartość lub typ, np.

```
functor f (type t) =
  struct
    fun f (x:t) = x
  end
```

Niekiedy bywa wygodne zdefiniowanie funktora bez parametrów postaci

$$\texttt{functor } f \texttt{ () = } S$$

Aplikacja funktora f do argumentu ma wówczas postać:

$$\texttt{structure } T \texttt{ = } f\texttt{()}$$

15.3.4. Deklaracje typów w strukturach

Na początku bieżącego podrozdziału zaznaczyliśmy, że nasz przykład modularyzacji programów za pomocą typów rekordowych daje jedynie intuicję, ale nie rozwiązuje problemu. Dotychczas przez analogię z rekordami opisaliśmy struktury, sygnatury i funktory, ale do tej pory nie oferowały one niczego nowego, przeciwnie, poza być może czytelniejszą składnią i ogólniejszym warunkiem dopasowania struktury do sygnatury (pomijanie komponentów), system modułów był istotnie słabszy (np. brak rekursji) od rekordów i nie oferował niczego nowego w zamian. Tą nowością jest możliwość umieszczania w strukturach deklaracji typów danych. Zatem struktura jest „rekordem", którego polami mogą być wartości, inne struktury *oraz typy*. Aby rekonstrukcję typów uczynić rozstrzygalną należało w zamian zabronić wielu rzeczy dopuszczalnych dla rekordów, np. definicji rekurencyjnych i dokonać „stratyfikacji" programu: na dolnym poziomie znajdują się wartości, które mogą zawierać jedynie inne wartości, na górnym — moduły, które mogą zawierać tak wartości, jak i inne moduły.

Wewnątrz struktury można umieszczać zarówno generatywne, jak i niegeneratywne deklaracje typów. Można więc użyć (generatywnej) deklaracji `datatype`:

```
- structure Nat1 =
=   struct
=     datatype nat = zero | succ of nat
=     fun add (succ n, m) = succ (add (n,m))
=       | add (zero, m) = m
=   end;
structure Nat1 :
  sig
    datatype nat = succ of nat | zero
    val add : nat * nat -> nat
  end
```

Dedukując najogólniejszą sygnaturę dla struktury `Nat` system ujawnił implementację typu `nat` i typ funkcji `add`. Jawnie sygnaturę struktury `Nat` zapiszemy w postaci:

```
signature NAT1 =
  sig
    datatype nat = succ of nat | zero
    val add : nat * nat -> nat
  end
```

Zatem specyfikacja `datatype` konkretnego typu danych ma dokładnie taką samą postać, jak deklaracja. Możemy jedynie zmienić kolejność konstruktorów. Nie możemy jednak żadnego z nich pominąć ani zmienić lub zawęzić jego typu. Typ jest widoczny we wszystkich specyfikacjach następujących w sygnaturze po nim. Można się też do niego i do jego konstruktorów odwoływać na zewnątrz struktury `Nat`:

```
- fun prod (Nat1.succ m, n) = Nat1.add (n, prod (m,n))
=   | prod (Nat1.zero, _) = Nat1.zero;
```

```
val prod = fn : Nat1.nat * Nat1.nat -> Nat1.nat
```

W strukturze może też wystąpić (niegeneratywna) deklaracja `type`:

```
- structure Nat2 =
=   struct
=     type nat = int
=     val zero = 0
=     fun succ n = n+1
=     fun add (m,n) = m+n
=   end;
structure Nat2 :
  sig
    type nat = int
    val add : int * int -> int
    val succ : int -> int
    val zero : int
  end
```

Tutaj specyfikacja `type` również jest takiej samej postaci, jak jego deklaracja. Jawnie sygnaturę struktury `nat` zapiszemy w postaci:

```
signature NAT2 =
  sig
    type nat = int
    val succ : nat -> nat
    val zero : nat
    val add : nat * nat -> nat
  end
```

Zauważmy, że dla większej czytelności w typach wartości `succ`, `zero` i `add` użyliśmy typu `nat` a nie `int`. Ponieważ jednak typ `nat` jest wyspecyfikowany jako równoważny z typem `int`, obie sygnatury są równoważne. Opisane wyżej dwie formy specyfikacji typu:

$$\texttt{datatype}\ \vec{\alpha}\ T\ \texttt{=}\ C_1\ \texttt{of}\ \sigma_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ \texttt{of}\ \sigma_n\ \texttt{|}\ D_1\ \texttt{|}\ \ldots\ \texttt{|}\ D_m$$
$$\texttt{type}\ \vec{\alpha}\ T\ \texttt{=}\ \sigma$$

są zwane *konkretnymi specyfikacjami typu*. Ujawniają one bowiem implementację tego typu w strukturze. W SML-u istnieje jeszcze trzecia specyfikacja typu, zwana *abstrakcyjną*, postaci

$$\texttt{type}\ \vec{\alpha}\ T$$

gdzie $\vec{\alpha}$ jest listą parametrów (zmiennych typowych), podobnie jak w deklaracji `type`. Za jej pomocą możemy (częściowo) ukryć implementację typu danych, np.

```
signature NAT3 =
  sig
    type nat
    val add : nat * nat -> nat
  end
```

Zawężenie sygnatury za pomocą NAT3 ukrywa konstruktory typu nat choć nazwa typu pozostaje widoczna:

```
- structure Nat3 : NAT3 = Nat1;
structure Nat3 : NAT3
- Nat3.zero;
Error: unbound variable or constructor: zero in path Nat3.zero
```

Konstruktory typu nat możemy na powrót ujawnić jako zwykłe wartości przez umieszczenie specyfikacji ich wartości w sygnaturze:

```
signature NAT4 =
  sig
    type nat
    val zero : nat
    val succ : nat -> nat
    val add  : nat * nat -> nat
  end
```

Tak wyspecyfikowane wartości zero i succ są dostępne:

```
- structure Nat4 : NAT4 = Nat1;
structure Nat4 : NAT4
- Nat4.add (Nat4.succ Nat4.zero, Nat4.zero);
val it = succ zero : Nat1.nat
```

ale nie mają już statusu konstruktorów:

```
- fun prod (Nat4.succ m, n) = Nat4.add (n, prod (m,n))
=   | prod (Nat4.zero, _) = Nat4.zero;
Error: variable found where constructor is required: Nat4.succ
Error: constant constructor applied to argument in pattern:bogus
Error: unbound variable or constructor: m
Error: variable found where constructor is required: Nat4.zero
```

Abstrakcyjna specyfikacja type czyniąc typ bardziej anonimowym pozwala na sterowanie stopniem abstrakcji. Jeżeli nasz program korzysta ze struktury o sygnaturze NAT4, to implementacja typu nat jest ukryta i możemy łatwo ją zmienić:

```
structure Nat5 : NAT4 =
  struct
```

```
    datatype nat = Nat of int
    val zero = Nat 0
    fun succ (Nat n) = Nat (n+1)
    fun add (Nat m, Nat n) = Nat (m+n)
  end
```

Zauważmy że całkowicie zmieniliśmy deklarację typu i funkcji na nim operujących. Struktura `Nat5` dopasowuje się do sygnatur `NAT3` i `NAT4`, ale nie dopasowuje się do sygnatur `NAT1` i `NAT2`.

15.3.5. Przezroczystość sygnatur

Informacja o typie zadanym abstrakcyjną specyfikacją `type` w sygnaturze „wycieka" na zewnątrz struktury. Dla przykładu system „wie", że wartość `Nat6.zero` jest tak na prawdę typu `int` i godzi się na obliczenie wyrażenia:

```
- structure Nat6 : NAT4 = Nat2
= Nat6.zero + Nat6.succ 5;
structure Nat6 : NAT4
val it = 6 : Nat2.nat
```

Jest to związane z faktem, że deklaracja `type` w strukturze `Nat6` nie jest generatywna — nie tworzy nowego typu. W standardzie SML-a przyjęto, że zawężenie sygnatury nie ukrywa „prawdziwej natury" typu i że kontrola typów przebiega po sięgnięciu do generatywnych deklaracji odpowiednich typów. Dlatego typy `Nat1.nat`, `Nat3.nat` i `Nat4.nat` są równoważne i są typem `Nat1.nat` (ponieważ jego definicja w strukturze `Nat1` jest generatywna). Typ `Nat5.nat` jest nowym typem, zaś typy `Nat2.nat` i `Nat6.nat` są równoważne z typem `int`. W szczególności możemy mieszać ze sobą wartości zdefiniowane w strukturach `Nat1`, `Nat3` i `Nat4`, np.

```
- Nat3.add (Nat1.succ Nat4.zero, Nat1.zero);
val it = succ zero : Nat1.nat
```

Jak się za chwilę przekonamy, mechanizm ujawniania informacji o typach zdefiniowanych w strukturze jest w pewnych sytuacjach bardzo wygodny, jednak nie zapewnia dostatecznego stopnia rozdzielenia specyfikacji i implementacji modułów. Dlatego w SML-u istnieją dwa rodzaje zawężenia sygnatury: *przezroczyste* (ang. *transparent*), które już znamy, postaci S `:` Σ i *nieprzezroczyste* (ang. *opaque*) postaci S `:>` Σ. Jeżeli napiszemy

```
structure Nat7 :> NAT4 = Nat2
```

wówczas żadna informacja o typach i wartościach zadeklarowanych w strukturze `Nat7` ponad to, co jest wyspecyfikowane w sygnaturze `NAT4` nie jest brana przez system pod uwagę. Dlatego system ujawnia naturę wartości `zero` w strukturze `Nat6`, zaś nie czyni tego w strukturze `Nat7`:

```
- Nat6.zero;
val it = 0 : Nat2.nat
- Nat7.zero;
val it = - : Nat7.nat
```

(znak - zamiast 0 w odpowiedzi systemu oznacza odmowę podania przez system wartości Nat7.zero). Co więcej, system „nie wie", że Nat7.zero jest liczbą 0 typu int i nie pozwala na użycie tej wartości w wyrażeniu typu int:

```
- Nat6.zero + 1;
val it = 1 : Nat2.nat
- Nat7.zero + 1;
Error: operator and operand don't agree [literal]
  operator domain: Nat7.nat * Nat7.nat
  operand:         Nat7.nat * int
  in expression:
    Nat7.zero + 1
```

Nieprzezroczystego zawężenia sygnatury można też używać do zawężenia sygnatury wyniku funktora:

$$\texttt{functor}\ f\ (s : \Sigma_s)\ \texttt{:>}\ \Sigma_T\ \texttt{=}\ T$$

Zauważmy natomiast, że zawężenie sygnatury w parametrach formalnych funktora działa zawsze tak, jakby było nieprzezroczyste, w treści funktora nie jest bowiem dostępna informacja o jego parametrach faktycznych. Dlatego inny sposób wymuszenia nieprzezroczystego dopasowania sygnatury, bardzo faworyzowany przez niektórych programistów, polega na definiowaniu jedynie funktorów, nawet jeśli funktorowi nie przekazujemy argumentów. Dzięki temu wszystkie wartości nielokalne, do których są odwołania w strukturze są parametrami formalnymi funktorów a nie istniejącymi strukturami i nie można wykorzystać żadnej dodatkowej informacji (nie istnieje ona bowiem) ponad to, co jest opisane w sygnaturze. Dla przykładu w strukturze Nat8 widać, że typ Nat8.one jest równoważny z int:

```
- structure Nat8 = struct val one = Nat6.zero + 1 end;
structure Nat8 : sig val one : Nat2.nat end
```

Jednak (na pierwszy rzut oka całkiem równoważna deklaracja)

```
- functor Nat9fun (N : NAT4) = struct val one = N.zero + 1 end
= structure Nat9 = Nat9fun (Nat6);
Error: operator and operand don't agree [literal]
  operator domain: N.nat * N.nat
  operand:         N.nat * int
  in expression:
    N.zero + 1
```

jest niepoprawna, ponieważ parametr N funktora jest jedynie parametrem *formalnym* i nie wiadomo, jak zostanie zaprogramowany argument *faktyczny* przekazany funktorowi podczas

tworzenia struktury `Nat9` i czemu będzie w nim równoważny typ `N.nat`. Dzięki temu w definicji funktora `Nat9fun` wymusiliśmy traktowanie typu `N.nat` jako typu abstrakcyjnego.

15.3.6. Specyfikacja exception

Ponieważ deklaracja `exception` wprowadza nowy konstruktor do (konkretnego) typu danych `exn`, spodziewamy się, że wyjątki można specyfikować w sposób podobny do specyfikacji `datatype`. Istotnie, w sygnaturach możemy umieścić *specyfikację* `exception` o składni identycznej z *deklaracją* `exception`, np.

```
- signature S = sig exception E end
= structure T : S = struct exception E end;
signature S = sig exception E end
structure T : S
- T.E;
val it = E(-) : exn
```

Deklaracja `exception` w strukturze dopasowuje się do specyfikacji `exception` w sygnaturze tylko wówczas, gdy obie mają dokładnie taką samą postać.

15.3.7. Specyfikacja typu równościowego

Nieprzezroczyste zawężenie sygnatury

$$\texttt{signature}\ \Sigma\ \texttt{=}\ \texttt{sig}\ \ldots\ \texttt{type}\ \vec{\alpha}\ T\ \ldots\ \texttt{end}$$

lub umieszczenie struktury o sygnaturze Σ jako parametru funktora powoduje traktowanie typu $\vec{\alpha}\ T$ jako typu całkowicie abstrakcyjnego o którym wiadomo jedynie, że ma parametry $\vec{\alpha}$ i że można na nim wykonywać operacje opisane w sygnaturze Σ. Nie można nawet porównywać elementów tego typu. Ponieważ w SML-u podzielono wszystkie typy na równościowe i nierównościowe, dobrze byłoby mieć sposób wyspecyfikowania, że $\vec{\alpha}\ T$ jest (abstrakcyjnym) typem równościowym. W tym celu w specyfikacji `type` zamiast słowa kluczowego `type` piszemy `eqtype`. Specyfikacja `eqtype` ma dokładnie takie samo znaczenie jak `type`, a ponadto pozwala używać operacji porównania dla elementów typu w niej wyspecyfikowanego, np.

```
functor F (S : sig type t val x : t and y : t end) =
  struct
    val b = S.x = S.y
  end
```

powoduje błąd kompilacji:

```
Error: operator and operand don't agree (equality type required)
  operator domain: ''Z * ''Z
  operand:         S.t * S.t
```

```
  in expression:
    = (x,y)
```

podczas gdy

```
functor F (S : sig eqtype t val x : t and y : t end) = ...
```

kompiluje się poprawnie. Podczas aplikacji funktora `F` do struktury `S` system sprawdzi, czy rzeczywiście typ `S.t` jest równościowy i jeśli nie — odmówi skompilowania takiej aplikacji. W Adzie mamy podobne specyfikacje:

```
type T is private
```

specyfikuje parametr rodzajowy `T`, który jest typem abstrakcyjnym, dla którego zdefiniowano tylko jedną operację — porównanie, podczas gdy

```
type T is limited private
```

jest całkowicie abstrakcyjnym typem dla którego nie zdefiniowano żadnych operacji. Pierwsza specyfikacja odpowiada SML-owej specyfikacji `eqtype`, podczas gdy druga — specyfikacji `type`.

Interpretacja elementów typu danych w pewnym zbiorze X jest *wolna*, jeśli różne wartości tego typu reprezentują różne elementy zbioru X. Ponieważ interpretacja elementów danego typu w programie jest ustalona, często o samym typie mówi się, czy jest wolny, czy nie. Podręcznikowy przykład typu, który nie jest wolny, to implementacja liczb całkowitych:

```
datatype positive = One | Succ of positive
datatype integer = Zero | Succ of integer | Pred of integer
```

Typ `positive` reprezentujący liczby dodatnie jest wolny, ponieważ różne wartości tego typu zawsze reprezentują różne liczby całkowite. Typ `integer` nie jest wolny, ponieważ nieskończenie wiele różnych wartości tego typu reprezentuje np. liczbę 0:

```
Zero
Succ (Pred Zero)
Pred (Pred (Succ (Succ Zero)))
```

itd. Wolność jest kwestią naszej interpretacji, a nie syntaktyczną własnością typu. Jeśli będziemy używać typu `integer` do reprezentowania liczb całkowitych, to standardowa relacja porównania nie będzie zgadzać się z naszą intencją. Dla takiego typu operację porównania należy zaprogramować samodzielnie. Syntaktycznie identyczny z typem `integer` typ

```
datatype walk = Center | Up of walk | Down of walk
```

którego wartości opisują położenie punktu podczas błądzenia po kracie (`Center` jest punktem wyjścia, `Up` oznacza, że przesuwamy się o jedno oczko kraty w górę i o jedno w prawo, a `Down` — o jedno w dół i o jedno w prawo) jest wolny. Typy `integer` i `walk` różnią się jedynie wyborem identyfikatorów, maszyna przetwarza je tak samo, tylko my, ludzie, rozumiemy je różnie.

Często wybór innej implementacji pozwala na zaprogramowanie typu wolnego, np.

```
datatype integer = Zero | Plus of positive | Minus of positive
```

Brak wolności pojawia się często wtedy, gdy reprezentujemy (multi-)zbiory elementów w postaci list, ponieważ porządek na liście jest istotny. Jeśli są to elementy typu wolnego liniowo uporządkowanego, to możemy je przechowywać na liście w kolejności np. rosnącej i wówczas niektóre wartości naszego typu nie będą w ogóle reprezentować niczego. Pozostały fragment będzie wolny. Tu jednak pojawia się niewykrywalny w czasie kompilacji problem sprawdzenia, czy wszystkie dane są sensowne (z podobną sytuacją można się spotkać implementując np. sterty lub drzewa poszukiwań, gdzie nie każda wartość danego typu jest sensowna).

Za pomocą specyfikacji `eqtype` należy specyfikować jedynie typy wolne, gdy jest naszą intencją, by użytkownik korzystał ze standardowej relacji porównania. Używając specyfikacji `type` możemy zabronić użytkownikowi porównywania elementów typu, który nie jest wolny.

15.3.8. Replikacja typów i wyjątków

Oto następujący problem: rozważmy sygnaturę:

```
signature S =
  sig
    datatype T = C | D
    val f : T -> int
  end
```

i strukturę o tej sygnaturze:

```
structure s1 : S =
  struct
    datatype T = C | D
    fun f C = 1
      | f D = 2
  end
```

Chcemy zdefiniować strukturę `s2` o tej samej sygnaturze, ale tak, by funkcja `s1.f` mogła też działać na danych z tej nowej struktury. Nie możemy ponownie użyć generatywnej deklaracji typu:

```
structure s2 : S =
  struct
    datatype T = C | D
    fun f C = 3
      | f D = 4
  end
```

ponieważ wówczas typy `s1.T` i `s2.T` są różne:

```
- s1.f (s2.C);
Error: operator and operand don't agree [tycon mismatch]
  operator domain: s1.T
  operand:         s2.T
  in expression:
    s1.f C
```

Nie możemy też użyć niegeneratywnej deklaracji `type`:

```
structure s2 : S =
  struct
    type T = s1.T
    fun f C = 3
      | f D = 4
  end
```

ponieważ wówczas konstruktory `C` i `D` są niedostępne i pojawia się błąd kompilacji:

```
Error: unmatched constructor specification: C
Error: unmatched constructor specification: D
```

Zadeklarowanie ich wartości także nie pomoże:

```
structure s2 : S =
  struct
    type T = s1.T
    val C = s1.C
    val D = s1.D
    fun f C = 3
      | f D = 4
  end
```

ponieważ dalej nie mają one statusu konstruktorów i dlatego nie można z nich korzystać we wzorcach. W takich sytuacjach można użyć deklaracji *replikacji typu* postaci:

$$\texttt{datatype}\ T_1\ \texttt{=}\ \texttt{datatype}\ T_2$$

gdzie T_1 jest identyfikatorem, zaś T_2 długim identyfikatorem. W naszym przykładzie możemy napisać

```
structure s2 : S =
  struct
    datatype T = datatype s1.T
    fun f C = 3
      | f D = 4
  end
```

Ponieważ deklaracja replikacji jest niegeneratywna, typy `s1.T` i `s2.T` są równe i system dopuszcza ich mieszanie w jednym wyrażeniu:

```
- s1.f (s2.C);
val it = 1 : int
```

Z podobnych powodów w celu umożliwienia importowania wyjątków z innych struktur, wprowadzono deklarację *replikacji wyjątku* postaci

$$\texttt{exception}\ E_1\ \texttt{=}\ E_2$$

gdzie E_1 i E_2 są dwiema nazwami wyjątków, przy czym E_2 może być nazwą długą.

15.3.9. Modularyzacja programu a przezroczystość sygnatur

Rozważmy moduł implementujący operacje symboliczne na wyrażeniach arytmetycznych opisany sygnaturą:

```
signature EXPR =
  sig
    datatype expr = Var of string | Real of real | ++ of expr * expr
    val diff : expr * string -> expr
  end
```

Zdecydowałem się potraktować typ `expr` jako konkretny typ danych i ujawnić jego implementację (inaczej nie moglibyśmy zbudować ani jednej wartości tego typu). Odpowiednią strukturę postanowiłem zaprogramować później.

Obecnie zamierzam zaimplementować operację konwersji abstrakcyjnych drzew rozbioru typu `expr` do łańcuchów znaków. Gdyby implementacja struktury `Expr : EXPR` już istniała, mógłbym napisać:

```
structure ExprPrint =
  struct
    fun toString (Expr.Var x) = x
      | toString (Expr.Real r) = Real.toString r
      | toString (Expr.++ (e1,e2)) =
          toString e1 ^ " + " ^ toString e2
  end
```

Ponieważ struktura `ExprPrint` korzysta ze struktury `Expr` o sygnaturze `EXPR`, której jeszcze nie zdefiniowałem, mogę strukturę `Expr` uczynić jej parametrem, tj. zaprogramować funktor:

```
functor ExprPrintFun (Expr : EXPR) =
  struct
    fun toString (Expr.Var x) = x
      | toString (Expr.Real r) = Real.toString r
```

```
      | toString (Expr.++ (e1,e2)) =
          toString e1 ^ " + " ^ toString e2
  end
```

Zauważmy, że funktor `ExprPrintFun` został zdefiniowany przed zaprogramowaniem struktury `Expr`, w szczególności przed zdefiniowaniem typu `expr` (który w chwili definiowania funktora *nie istnieje*). Aby móc w podobny sposób korzystać ze struktury `ExprPrint` przed jej zdefiniowaniem, a nawet przed zdefiniowaniem funktora `ExprPrintFun`, wystarczy napisać jej sygnaturę. Tu jednak czeka nas niespodzianka: typ `expr` jest zdefiniowany w strukturze `Expr`, jest więc parametrem funktora `ExprPrintFun` i jeszcze go nie ma, nie możemy się zatem do niego odwoływać. Wyjściem jest zdefiniowanie drugiego typu, równoważnego z typem `expr` wewnątrz struktury `ExprPrint`:

```
signature EXPR_PRINT =
  sig
    type expr
    val toString : expr -> string
  end
```

Obecnie musimy w definicji funktora `ExprPrintFun` umieścić definicję typu `expr` odpowiadającą powyższej specyfikacji. Nie może to być definicja generatywna, wówczas bowiem oba typy byłyby różne i nie można byłoby używać funkcji `ExprPrint.toString` do wartości typu `Expr.expr`. Dlatego wybierzemy deklarację replikacji typu:

```
functor ExprPrintFun (Expr : EXPR) : EXPR_PRINT =
  struct
    datatype expr = datatype Expr.expr
    infix 6 ++
    fun toString (Var x) = x
      | toString (Real r) = Real.toString r
      | toString (e1 ++ e2) =
          toString e1 ^ " + " ^ toString e2
  end
```

Najwyższy już czas zdefiniować strukturę `Expr`:

```
structure Expr : EXPR =
  struct
    datatype expr = Var of string | Real of real | ++ of expr * expr
    infix 6 ++
    infix 9 diff
    fun (Var x) diff y = Real (if x=y then 1.0 else 0.0)
      | (Real _) diff _ = Real 0.0
      | (e1 ++ e2) diff x = e1 diff x ++ e2 diff x
 end
```

Rodzajowego modułu `ExprPrintFun` nie musimy powtórnie kompilować. Wystarczy zdefiniować strukturę `ExprPrint` jako instancję powyższego modułu *nastrojoną* na implementację modułu `Expr`:

```
- structure ExprPrint = ExprPrintFun (Expr);
structure ExprPrint : EXPR_PRINT
```

Dzięki przezroczystości sygnatur system jest w stanie stwierdzić, że typy `ExprPrint.expr` oraz `Expr.expr` są równoważne i pozwala napisać:

```
- ExprPrint.toString (Expr.++ (Expr.Real 3.4, Expr.Var "z"));
val it = "3.4 + x" : string
```

Obliczenie powyższego wyrażenia jest możliwe jedynie dzięki temu, że sygnatura struktury `ExprPrint` jest przezroczysta. Zadeklarowanie struktury `ExprPrint` z nieprzezroczystym zawężeniem sygnatury wyklucza taką możliwość:

```
- structure ExprPrint :> EXPR_PRINT = ExprPrintFun (Expr);
structure ExprPrint : EXPR_PRINT
- ExprPrint.toString (Expr.++ (Expr.Real 3.4, Expr.Var "z"));
Error: operator and operand don't agree [tycon mismatch]
  operator domain: ExprPrint.expr
  operand:         Expr.expr
  in expression:
    ExprPrint.toString (++ (Real 3.4,Var "z"))
```

15.3.10. Specyfikacja sharing

Z przezroczystości nie możemy skorzystać wówczas, gdy struktury `Expr` oraz `ExprPrint` obie są parametrami kolejnego funktora, implementującego np. program główny:

```
- functor MainFun (structure Expr : EXPR
=                 structure ExprPrint : EXPR_PRINT) =
=   struct
=     fun main () = ExprPrint.toString (Expr.Var "z")
=   end;
Error: operator and operand don't agree [tycon mismatch]
  operator domain: ?.expr
  operand:         Expr.expr
  in expression:
    ExprPrint.toString (Var "z")
```

Istotnie, nigdzie nie obiecaliśmy, że implementacje struktur `Expr` i `ExprPrint` będą współdzielić ten sam typ `expr`! Dlatego w sygnaturach możemy używać specjalnego rodzaju specyfikacji, tzw. *specyfikacji* `sharing` postaci:

$$\texttt{sharing type } T_1 \texttt{ = } T_2$$

gdzie T_1 i T_2 są długimi identyfikatorami typów. Typy T_1 i T_2 muszą pojawić się w tej samej sygnaturze we wcześniejszych specyfikacjach. Specyfikacja `sharing` deklaruje syntaktyczną równoważność nazw (tj. identyfikatorów), a nie semantyczną równość typów, nie można np. napisać

```
sharing type S.t1 = S.t2 * S.t2
```

ponieważ po prawej stronie znaku równości występuje nie nazwa typu, tylko wyrażenie typowe. Nie można też napisać

```
sharing type S.t = int
```

gdyż `int` nie jest typem lokalnie wyspecyfikowanym w sygnaturze.

W naszym przykładzie aby „obiecać", że typy `Expr.expr` i `ExprPrint.expr` są równoważne, należy napisać:

```
sharing type ExprPrint.expr = Expr.expr
```

w sygnaturze (anonimowej) struktury będącej parametrem funktora `MainFun`:

```
functor MainFun (structure Expr : EXPR
                 structure ExprPrint : EXPR_PRINT
                 sharing type ExprPrint.expr = Expr.expr) =
  struct
    fun main () = ExprPrint.toString (Expr.Var "z")
  end;
```

Podczas aplikacji funktora `MainFun` do argumentu system sprawdzi zgodność typów i nie przyjmie takich struktur `Expr` i `ExprPrint`, w których typy `expr` zostały różnie zdefiniowane jako parametrów funktora `MainFun`.

Dodatkową formą specyfikacji `sharing` jest

$$\texttt{sharing } S_1 \texttt{ = } S_2$$

gdzie S_1 i S_2 są nazwami struktur. Jest ona równoważna ciągowi specyfikacji

$$\texttt{sharing type } S_1.t_i \texttt{ = } S_2.t_i$$

dla wszystkich typów t_i występujących w strukturach S_1 i S_2. Ta forma specyfikacji jest jednak rzadko używana.

15.3.11. Wyrażenie with type

Specyfikacja `sharing` nie pozwala na wymuszenie równości typu pojawiającego się w sygnaturze i typu opisanego np. wyrażeniem typowym, np. w kontekście deklaracji

```
signature NAT =
  sig
    type nat
    val zero : nat
    val succ : nat -> nat
    val add  : nat * nat -> nat
  end
```

nie możemy napisać fragmentu specyfikacji postaci

```
structure N : NAT
sharing type N.nat = int
```

Aby narzucić dodatkowe warunki na typ `nat` z sygnatury `NAT` możemy sygnaturę struktury `N` opisać za pomocą *wyrażenia* `with type`, którego wartością jest sygnatura, postaci:

```
structure N : NAT where type nat = int
```

Wyrażenie `where type` ma postać:

$$\Sigma \texttt{ where type } T \texttt{ = } \sigma$$

i opisuje sygnaturę zgodną z sygnaturą Σ w której dodatkowo typ T jest równy typowi σ.

15.3.12. Zasada zamkniętości sygnatur

Podstawową zasadą dobrego stylu w programowaniu w SML-u jest *zasada zamkniętości sygnatur* (*signature closure rule*) według której jedynymi wolnymi identyfikatorami w sygnaturze powinny być nazwy predefiniowanych obiektów standardowych (np. `int`, `'a list` itp.) i innych sygnatur.

Niektórzy programują moduły wykorzystując jedynie funktory nawet wówczas, gdy definiowana struktura nie ma parametrów. Kompilacja funktora nie wprowadza do środowiska żadnego nowego obiektu, wszystkie funktory są zatem kompilowane w środowisku standardowym i nie mogą korzystać z żadnych (poza standardowymi) obiektów nielokalnych. Kolejność kompilacji funktorów jest nieistotna. Dopiero ostatnia faza budowy programu — łączenie modułów polega na aplikacji funktorów do argumentów. Budowa programu modularnego jest zatem rozbita na trzy etapy:

1. Specyfikacja modułów za pomocą zestawu zdefiniowanych globalnie sygnatur, do których wszystkie moduły (i np. wszyscy programiści) mogą się odwoływać:

   ```
   signature EXPR =
     sig
       datatype expr = Var of string | Real of real |
                     ++ of expr * expr
       val diff : expr * string -> expr
   ```

```
  end
signature EXPR_PRINT =
  sig
    type expr
    val toString : expr -> string
  end
signature MAIN =
  sig
    val main : unit -> string
  end
```

Sygnatury odwołują się nielokalnie jedynie do obiektów standardowych (tutaj `string` i `real`).

2. Implementacja funktorów odwołujących się jedynie do obiektów standardowych i zdefiniowanych w poprzedniej fazie sygnatur:

```
functor ExprPrintFun (Expr : EXPR) : EXPR_PRINT =
  struct
    type expr = Expr.expr
    fun toString (Expr.Var x) = x
      | toString (Expr.Real r) = Real.toString r
      | toString (Expr.++ (e1,e2)) =
          toString e1 ^ " + " ^ toString e2
  end
functor ExprFun () : EXPR =
  struct
    datatype expr = Var of string | Real of real |
                    ++ of expr * expr
    infix 6 ++
    infix 9 diff
    fun (Var x) diff y = Real (if x=y then 1.0 else 0.0)
      | (Real _) diff _ = Real 0.0
      | (e1 ++ e2) diff x = e1 diff x ++ e2 diff x
  end
functor MainFun (structure Expr : EXPR
                 structure ExprPrint : EXPR_PRINT
                 sharing type ExprPrint.expr = Expr.expr)
        : MAIN =
  struct
    fun main () = ExprPrint.toString (Expr.Var "z")
  end
```

Kolejność definiowania i kompilacji funktorów jest całkowicie dowolna i są one od

siebie niezależne. Po ich skompilowaniu środowisko pozostaje standardowe. Nie utworzono jeszcze żadnej struktury.

3. Łączenie modułów w program. W tym miejscu bardzo ważna jest kolejność tworzenia struktur:

```
structure Main =
  let
    structure Expr = ExprFun()
  in
    MainFun (structure Expr = Expr
             structure ExprPrint = ExprPrintFun (Expr))
  end
```

Do środowiska wstawiono jedynie strukturę `Main` implementującą główny program. Należy uważać, by nie stało się tak, że dwie struktury *wzajemnie* od siebie zależą. Ze względu na brak rekurencyjnych definicji struktur, łączenie programu będzie w takim przypadku niemożliwe.

Zmiany w pierwszej fazie (definiowanie sygnatur) mogą pociągać konieczność przebudowywania całego programu. Zmiany w implementacjach funktorów są całkowicie lokalne — wystarczy jeszcze raz przekompilować dany funktor i wykonać (zwykle krótką) fazę łączenia, by zbudować cały program na nowo. Kompilowanie pozostałych modułów nie jest konieczne.

15.3.13. Dyrektywa open

W standardzie języka wprowadzono *dyrektywę* `open` o składni

$$\texttt{open}\ S_1\ \ldots\ S_n$$

gdzie $S_1 \ldots S_n$ są (być może długimi) nazwami struktur. Po wydaniu tej dyrektywy, do komponentów struktur S_i możemy się odwoływać bez konieczności prefiksowania ich nazw nazwami struktur S_i, np:

```
- structure S = struct val x = 5 end;
structure S : sig val x : int end
- open S;
open S
val x = 5 : int
- x;
val it = 5 : int
- S.x;
val it = 5 : int
```

Po otwarciu struktury S identyfikator x został wstawiony do środowiska. Do wartości x możemy się oczywiście też odwołać za pomocą długiego identyfikatora S.x. Jeżeli następnie przesłonimy wartość x kolejną deklaracją, pozostaje ona widoczna na powrót jedynie w strukturze S:

```
- val x = 7;
val x = 7 : int
- S.x;
val it = 5 : int
```

Dyrektywa open przypomina przypomina nieco instrukcję with Pascala. Tu jednak otwarcie następuje permanentnie. Ze względu na niekontrolowane odsłanianie identyfikatorów z deklaracji open należy korzystać z najwyższą ostrożnością. Niekiedy jednak z pewnych modułów korzystamy tak często, że używanie długich identyfikatorów jest bardzo niepraktyczne[2] i strukturę należy otworzyć. Najlepiej zrobić to jednak w kontekście lokalnym tj. local lub let

```
- let open S in x + x + x + x + x end;
val it = 25 : int
```

ewentualnie wewnątrz kontrolowanej zawężeniem struktury. Zawężenie jest konieczne, ponieważ otwarcie struktury wewnątrz innej struktury sprawia, że wszystkie ujawnione obiekty są dołączone jako komponenty domniemanej sygnatury tej struktury:

```
- structure T = struct open S end;
structure T : sig val x : int end
```

15.3.14. Dyrektywa include

Co prawda sygnatury możemy definiować niezależnie, niekiedy jednak, szczególnie w sytuacjach, gdy moduł $S_2 : \Sigma_2$ jest rozszerzeniem modułu $S_1 : \Sigma_1$, wszystkie specyfikacje umieszczone w sygnaturze Σ_1 trzeba by przepisać w sygnaturze Σ_2. Aby tego uniknąć wprowadzono specjalną *dyrektywę* include postaci

$$\texttt{incude } \Sigma_1 \ \ldots \ \Sigma_n$$

która może pojawić się wewnątrz sygnatury i ma, z grubsza mówiąc, takie samo działanie, jak fizyczne wstawienie tekstu sygnatur $\Sigma_1 \ldots \Sigma_n$ do wnętrza sygnatury w której występuje, np.:

```
- signature SIG1 = sig type t end
= signature SIG2 = sig include SIG1 val x : t end;
signature SIG1 = sig type t end
signature SIG2 =
```

[2] Podstawowy problem polega na komplikacjach użycia identyfikatorów infiksowych w strukturach.

```
  sig
    type t
    val x : t
  end
```

Ze względu na nielokalność dyrektywa `include` ma swoich zagorzałych przeciwników argumentujących, że jest równie niebezpieczna i nieelegancka jak `open`. Z drugiej strony fanatycy programowania obiektowego widzą w niej pewną poronną formę dziedziczenia i używają jej namiętnie. Dziedziczenie dużo „czyściej" można zrealizować za pomocą lokalnej deklaracji struktury:

```
- signature SIG1 = sig type t end
= signature SIG2 = sig structure S1 : SIG1 val x : S1.t end;
signature SIG1 = sig type t end
signature SIG2 =
  sig
    structure S1 : sig type t end
    val x : S1.t
  end
```

Jedyną wadą (a może zaletą?) takiego rozwiązania jest pojawienie się długich identyfikatorów.

15.3.15. Dyrektywa infix w strukturach

Zakres obowiązywania dyrektywy `infix` jest ograniczany przez deklaracje `local` i wyrażenia `let`. Dyrektywa `infix` umieszczona wewnątrz definicji struktury zmienia łączliwość identyfikatora jedynie w jej wnętrzu. Zatem dyrektywa `infix` jest znaczeniowo związana z miejscem *użycia* identyfikatora, nie z miejscem jego *zadeklarowania*.

Długie identyfikatory zawsze są prefiksowe i nie można zmienić ich łączliwości. Jest to niekiedy bardzo kłopotliwe. Jeśli bowiem zdefiniujemy strukturę zawierającą (domyślnie) identyfikator infiksowy, np. `++`:

```
structure S =
  struct
    infix 7 ++
    exception VecSum
    fun (x::xs) ++ (y::ys) = x+y :: xs++ys : int list
      | nil ++ nil = nil
      | _ ++ _ = raise VecSum
  end
```

to jest on infiksowy jedynie wewnątrz struktury `S`. Na zewnątrz musimy pisać `S.++(xs,ys)` a wyrażenie `xs S.++ ys` oraz dyrektywa `infix S.++` są niepoprawne. Standard przewiduje, że wykonanie dyrektywy `open S` nie powoduje wykonania zawartych w strukturze

S dyrektyw `infix`. Zatem zgodnie ze standardem, w celu użycia funkcji ++ w sposób infiksowy, należałoby np. otworzyć S a następnie powtórnie wydać dyrektywę `infix 7 ++`. W niezgodzie ze standardem, kompilator SML/NJ wykonuje (potajemnie) dyrektywę `infix` zawartą w strukturze S podczas otwierania tej struktury, np.

```
- nonfix ++;
nonfix ++
- open S;
open S
exception VecSum = VecSum
val ++ = fn : int list * int list -> int list
- [1,2] ++ [3,4];
val it = [4,6] : int list
```

Jest to znaczne ułatwienie i pułapka zarazem — przy otwieraniu struktury nie wiemy jak zmieni się składnia języka! Niektóre identyfikatory mogą nieoczekiwanie stać się np. infiksowe. Takie rozszerzenie powinno być skojarzone ze *specyfikacją* `infix` w sygnaturze. SML/NJ pozwala na umieszczenie takiej specyfikacji, ale ostrzega, że ją zignoruje:

```
- signature SIG =
  sig
    infix 7 ++
    exception VecSum
    val ++ : int list * int list -> int list
  end;
Warning: Fixity specification in signatures are ignored
```

i nie sprawdza podczas zawężania sygnatury, czy w strukturze występuje odpowiadająca jej dyrektywa `infix`. Mimo to warto umieszczać specyfikację `infix` ze względów informacyjnych.

Zamiast otwierać strukturę w celu umożliwienia korzystania z operatorów infiksowych, często łatwo sobie poradzić w inny sposób — deklarujemy nową, lokalną wartość i zmieniamy lokalnie jej łączliwość:

```
let
  val ++ = S.++
  infix 7 ++
in
  [1,2] ++ [3,4]
end
```

Powyższe rozwiązanie działa wyśmienicie dopóty, dopóki identyfikatorem infiksowym nie jest konstruktor. W przeciwnym razie nowego identyfikatora nie możemy używać we wzorcach. W takich wypadkach może pomóc deklaracja replikacji typu.

15.3.16. Polimorfizm w modułach

W Core ML-u powszechnie używa się polimorfizmu parametrycznego, osiąganego przez niedospecyfikowanie wartości, np. w funkcji identyczności:

```
fun id x = x
```

Ponieważ niczego nie możemy powiedzieć o naturze parametru `x` kompilator przyjmuje, że `x` może mieć *dowolny typ* i wnioskuje dla funkcji `id` typ $\alpha \rightarrow \alpha$, gdzie α jest *zmienną* przyjmującą jako wartość *dowolny* typ (a właściwie *schemat typu* $\forall\alpha.(\alpha \rightarrow \alpha)$, gdzie $\forall$ gwarantuje, że α jest dowolne). Zatem α jest tu niejako parametrem, za który będzie podstawiony odpowiedni typ w chwili użycia funkcji `id`, np. w wyrażeniu `id 5` zmienna α przyjmuje wartość `int`. Zgodnie z ideą niepodawania typów w programach, wszystko to dzieje się „za kulisami kompilacji" i o ile nie nastąpi błąd typowania, programista nie jest ani uczestnikiem, ani nawet świadkiem tego procesu.

Funktory wprowadzają podobny rodzaj polimorfizmu, w którym jednak typy muszą być podawane *explicite*:

```
functor idM (type T) = struct fun f (x:T) = x end
```

15.3.17. Precyzyjna kontrola polimorfizmu

Binarne drzewa poszukiwań można zaprogramować w strukturze o sygnaturze

```
signature BST1 =
  sig
    infix 5 $
    datatype 'a tree = $ of 'a * ('a tree * 'a tree) | %
    val insert : ('a * 'a -> bool) -> 'a * 'a tree -> 'a tree
    val delete : ('a * 'a -> bool) -> 'a * 'a tree -> 'a tree
    val maxel  : 'a tree -> 'a * 'a tree
    val find   : ('a * 'a -> bool) -> 'a tree -> 'a
  end
```

Na barkach użytkownika spoczywa tu obowiązek pilnowania, czy powyższe operacje są wykorzystywane zgodnie z ich przeznaczeniem, np. czy budując drzewo użytkownik za każdym razem korzysta z tej samej operacji porównania. W odróżnieniu od pojedynczych drzew i elementów do nich wstawianych, operacja porównania powinna być jedna, podobnie jak typ wszystkich elementów wstawianych do tego samego drzewa. Niewygodnie byłoby zaprogramować operacje na drzewach w postaci funkcji monomorficznych, bo trzeba by pisać ich implementacje osobno dla każdego typu elementów i dla każdej operacji porównania. Zamiast tego możemy typ i operację porównania uczynić parametrem funktora implementującego naszą bibliotekę operacji na drzewach. Elementy wstawiane do drzewa powinny być uporządkowane relacją <, piszemy więc najpierw sygnaturę zbioru uporządkowanego:

```
signature ORD_SET =
  sig
    type elem
    val < : elem * elem -> bool
  end
```

i sygnaturę naszej biblioteki:

```
signature BST =
  sig
    structure Label : ORD_SET
    infix 5 $
    datatype tree = $ of Label.elem * (tree * tree) | %
    val insert : Label.elem * tree -> tree
    val delete : Label.elem * tree -> tree
    val maxel  : tree -> Label.elem * tree
    val find   : tree -> Label.elem
  end
```

Następnie implementujemy moduł w postaci funktora

```
functor Bst (Label : ORD_SET) : BST =
  struct
    structure Label = Label
    infix 5 $
    datatype tree = $ of Label.elem * (tree * tree) | %
    ...
  end
```

Operacje na drzewach implementujemy tylko raz i pozostają one polimorficzne, teraz jednak w innym sensie: ograniczyliśmy swobodę użytkownika. Musi on zbudować tyle różnych struktur, ile typów i operacji porównania będzie wykorzystywał w swoim programie. System gwarantuje, że o ile struktura zostanie dobrze zbudowana (tj. `<` w definicji `Label` będzie relacją porządku), to budowane drzewa na pewno będą poprawnie zbudowanymi drzewami poszukiwań.

Rozdział 16

Abstrakcyjne typy danych

Do uzupełnienia: Konkretne i abstrakcyjne typy danych. Deklaracja `abstype` w SML-u. Wykorzystanie systemu modułów SML-a do definiowania typów abstrakcyjnych. Esej o przewadze Linux-a nad Windows-ami.

Rozdział 17

Programowanie obiektowe

Do uzupełnienia: Idea programowania obiektowego. Obiekty i klasy, dziedziczenie, `self` i `super`. Simula 67, C++, Java. Języki bezklasowe, Smalltalk.

Rozdział 18

Programowanie współbieżne

18.1. Modele współbieżności

Do uzupełnienia:

18.1.1. Procesy, zdarzenia i przeplot

18.1.2. Temporalny rachunek zdań z czasem liniowym

Rozważmy nieskończony zbiór *zmiennych zdaniowych* $P = \{p, q, r, \ldots\}$. Formuły *temporalnego rachunku zdań* budujemy ze zmiennych zdaniowych, spójników *prawdziwościowych*: negacji $\neg$, koniunkcji $\wedge$, alternatywy $\vee$, implikacji $\supset$ i równoważności $\leftrightarrow$, oraz *modalnych* spójników konieczności $\Box$ i możliwości $\Diamond$, używając nawiasów (i) w celu wymuszenia właściwego rozbioru gramatycznego (zakładamy następujące pierwszeństwo operatorów: najsilniej wiążą spójniki jednoargumentowe $\neg$, $\Box$ i $\Diamond$, następnie, w kolejności malejącej siły: $\wedge$, $\vee$, $\supset$ i $\leftrightarrow$; spójnik implikacji wiąże w prawo, pozostałe zaś spójniki binarne — w lewo). „Zwykłe” spójniki logiczne nazywa się *prawdziwościowymi*, ponieważ wartość logiczna zdania zawierającego taki spójnik jest zdeterminowana poprzez wartości logiczne składowych tego zdania. Spójniki *nieprawdziwościowe* nazywa się *modalnymi*. Prawdziwość zdania zawierającego takie spójniki nie jest lub nie jest w całości określona poprzez wartości logiczne jego składowych.

Zbiór formuł temporalnego rachunku zdań zawiera jako podzbiór zbiór formuł *klasycznego rachunku zdań* (CPC, *classical propositional calculus*). Jest to zbiór formuł nie zawierających modalnych spójników $\Box$ i $\Diamond$. Przypomnijmy, że formuła CPC jest *prawdziwa* (jest *tautologią* CPC), gdy jest spełniona w każdym modelu, tj. dla każdego wartościowania zmiennych. *Wartościowanie zmiennych* w CPC, to dowolne odwzorowanie $I : P \rightarrow \{0, 1\}$. Definicja spełniania formuły przy danym wartościowaniu jest następująca (zob. [173], str. 113):

$$
\begin{array}{ll}
I \models p & \text{jeśli } I(p) = 1 \\
I \models \neg\phi & \text{jeśli nie prawda, że } I \models \phi \\
I \models \phi \wedge \psi & \text{jeśli } I \models \phi \text{ oraz } I \models \psi \\
I \models \phi \vee \psi & \text{jeśli } I \models \phi \text{ lub } I \models \psi \\
I \models \phi \supset \psi & \text{jeśli } I \models \psi \text{ lub nie prawda, że } I \models \phi \\
I \models \phi \leftrightarrow \psi & \text{jeśli } I \models \phi \text{ wtedy i tylko wtedy, gdy } I \models \psi
\end{array}
$$

Formuła ϕ jest *prawdziwa*, oznaczenie: CPC $\models \phi$, jeśli jest prawdziwa dla każdego wartościowania $I : P \to \{0, 1\}$.

Obecnie zmierzamy do rozszerzenia pojęcia spełniania tak, by uchwycić zjawiska zmienne w czasie. Przyjmijmy, że chwile czasu będą reprezentowane przez liczby naturalne (zbiór liczb naturalnych oznaczamy ω). Zatem czas będzie *dyskretny*, *liniowy* i będzie miał *początek*.[1] Obecnie wartościowanie będzie funkcją *dwóch* parametrów: zmiennej zdaniowej i czasu, $I : P \times \omega \to \{0, 1\}$. Gdy $I(p, t) = 1$, mówimy, że przy wartościowaniu I zmienna zdaniowa p ma wartość logiczną 1 w chwili t. Podobnie jak w przypadku CPC, możemy wartościowanie zmiennych rozszerzyć do wartościowania formuł, otrzymując pojęcie spełniania w chwili t formuły ϕ przez wartościowanie I w PLTL (*propositional linear temporal logic*):

$$
\begin{array}{ll}
I, t \models p & \text{jeśli } I(p, t) = 1 \\
I, t \models \neg\phi & \text{jeśli nie prawda, że } I, t \models \phi \\
I, t \models \phi \wedge \psi & \text{jeśli } I, t \models \phi \text{ oraz } I, t \models \psi \\
I, t \models \phi \vee \psi & \text{jeśli } I, t \models \phi \text{ lub } I, t \models \psi \\
I, t \models \phi \supset \psi & \text{jeśli } I, t \models \psi \text{ lub nie prawda, że } I, t \models \phi \\
I, t \models \phi \leftrightarrow \psi & \text{jeśli } I, t \models \phi \text{ wtedy i tylko wtedy, gdy } I, t \models \psi
\end{array}
$$

Spójnikom modalnym przypisujemy następujące znaczenie: samo ϕ oznacza, że ϕ zachodzi *teraz*, $\Box\phi$ oznacza, że ϕ zachodzi *zawsze* (od teraz), zaś $\Diamond\phi$ oznacza, że ϕ zachodzi *niekiedy*, *czasem* (tj. że zdarzy się taki moment, że ϕ będzie spełniona):

$$
\begin{array}{ll}
I, t \models \Box\phi & \text{jeśli } I, t' \models \phi \text{ dla każdego } t' \geq t \\
I, t \models \Diamond\phi & \text{jeśli } I, t' \models \phi \text{ dla pewnego } t' \geq t
\end{array}
$$

Formuła ϕ jest *prawdziwa w PLTL*, oznaczenie: PLTL $\models \phi$, jeśli jest spełniona *zawsze* i *wszędzie*, tj. gdy $I, t \models \phi$ dla każdej interpretacji $I : P \times \omega \to \{0, 1\}$ i każdej chwili $t \in \omega$.

Uwagi. Logiki temporalne są zwykle traktowane jako szczególne przypadki tzw. *logik modalnych*. Współcześnie pojęcie modalności (tzw. *aletycznej*, tj. odnoszącej się do prawdziwości) wprowadził J. C. Lewis po to, by uporać się z paradoksami implikacji materialnej. Otóż zwykła implikacja (zwana materialną) $\phi \supset \psi$ jest prawdziwa, gdy ϕ jest fałszywe lub ψ prawdziwe. W szczególności musimy uznać — wbrew zdrowemu rozsądkowi — za prawdziwe wszelkie kontrfaktyczne zdania warunkowe, np. „Jeśli Kowalski ma dużego fiata, to wczoraj padało," w sytuacji, gdy Kowalski ma malucha, a wczoraj była piękna pogoda. Wolelibyśmy, by okres warunkowy wyrażał *konieczność* zajścia konkluzji, jeśli prawdziwe są przesłanki. Wprowadzamy spójniki modalne $\Box$ i $\Diamond$ o następującym znaczeniu: sama formuła ϕ oznacza, że ϕ jest *faktycznie prawdziwe*, $\Box\phi$ oznacza, że ϕ jest *konieczne* (zachodzi w każdym świecie, w szczególności tym faktycznym), zaś $\Diamond\phi$, że ϕ jest *możliwe* (zachodzi w pewnym świecie, niekoniecznie

[1] Nie jest to jedyne możliwe rozwiązanie. Rozważa się również czas, który nie jest liniowo uporządkowany (tzw. *branching-time logic*).

tym faktycznym). Spójniki te zachowują się bardzo podobnie, jak ich odpowiedniki temporalne, są np. wzajemnie dualne: nie prawda, że coś jest konieczne znaczy ni mniej, ni więcej tylko tyle, że możliwe jest, że to coś nie jest prawdziwe. Pomiędzy logiką modalną i temporalną są jednak pewne różnice, których tu jednak nie zamierzamy precyzować. Wyposażeni w aparat do „gdybania" możemy teraz zdefiniować spójnik tzw. implikacji ścisłej: $\phi \Rightarrow \psi \iff \neg\Box(\phi \wedge \neg\psi)$ (*niemożliwe jest*, by jednocześnie ϕ było prawdziwe i ψ fałszywe). Zauważmy przy tym, że dla implikacji materialnej $\phi \supset \psi \iff \neg(\phi \wedge \neg\psi)$ (*nieprawdziwe jest*, że ϕ jest prawdziwe i ψ fałszywe). Implikacja ścisła wyraża zatem konieczność implikacji materialnej: $\phi \Rightarrow \psi \iff \Box(\phi \supset \psi)$. Ponieważ jesteśmy skłonni przyjąć $\Box\phi \supset \phi$ za aksjomat naszej logiki, zatem jeśli zachodzi implikacja ścisła, zachodzi też i materialna, niekoniecznie jednak na odwrót. Jesteśmy w stanie wyobrazić sobie świat, w którym Kowalski istotnie ma dużego fiata i od kilku dni jest piękna pogoda. Zatem wczorajszy opad atmosferyczny nie jest warunkiem koniecznym zmotoryzowania Kowalskiego i cytowane wyżej zdanie w sensie implikacji ścisłej nie zachodzi.

Do uzupełnienia:

18.2. Programowanie współbieżne — programowanie w środowisku z pamięcią dzieloną
18.2.1. Semafory
18.2.2. Monitory
18.2.3. Warunkowe rejony krytyczne
18.3. Programowanie rozproszone
18.3.1. Ada
18.3.2. Concurrent ML
18.3.3. Occam
18.4. Współbieżność w języku C — semafory i sygnały

Literatura zastępcza: [19, 18, 185, 143, 67, 77, 76, 146]

18.5. Zadania

Zadanie 18.1. Pokaż, że PLTL jest *konserwatywnym rozszerzeniem* CPC, tj. formuła nie zawierająca spójników modalnych jest prawdziwa w PLTL wtedy i tylko wtedy, gdy jest prawdziwa w CPC.

Zadanie 18.2. Pokaż, że spójniki modalne $\Box$ i $\Diamond$ są *dualne* w podobny sposób jak kwantyfikatory $\forall$ i $\exists$, tj. że $\neg\Box\phi \leftrightarrow \Diamond\neg\phi$ (nie zawsze ϕ jest równoważne możliwe, że nie ϕ). Jakie inne podobieństwa zachodzą między spójnikami modalnymi i kwantyfikatorami? Co oznaczają formuły $\Box\Diamond\phi$ i $\Diamond\Box\phi$?

Zadanie 18.3. Na zbiorze formuł PLTL definiujemy relacje

$$\phi \sim \psi \iff \text{PLTL} \models \phi \leftrightarrow \psi$$
$$\phi \leq \psi \iff \text{PLTL} \models \phi \supset \psi$$

Pokaż że $\sim$ jest relacją równoważności a $\leq$ praporządkiem. Pokaż, że relacja

$$[\phi]_\sim \leq [\psi]_\sim \iff \phi \leq \psi$$

jest poprawnie zdefiniowana (definicja nie zależy od wyboru reprezentantów klas abstrakcji) i że jest porządkiem na zbiorze klas abstrakcji.

Rozważmy formuły zbudowane z jednej ustalonej zmiennej zdaniowej p z pomocą jedynie unarnych spójników $\neg$, $\Box$ i $\Diamond$. Ile klas abstrakcji ma relacja $\sim$ na zbiorze takich formuł? Wypisz ich najkrótszych reprezentantów. Narysuj diagram porządku $\leq$ na zbiorze tych reprezentantów.

Zadanie 18.4. Składnię formuł rozszerzamy o unarny spójnik X (jak neXt) i binarny spójnik U (jak Until) o następującym znaczeniu: $X\phi$, jeśli ϕ jest prawdziwa w następnej chwili (czas jest dyskretny), oraz $\phi\, U\, \psi$ gdy ϕ jest prawdziwe tak długo, nim zajdzie ψ. Zdefiniuj formalnie spełnialność powyższych formuł.

Zadanie 18.5. Rozważmy model, w którym czas jest pewnym zbiorem T uporządkowanym częściowo pewną relacją porządku $\leq$, niekoniecznie liniową. Napisz przykład formuły prawdziwej w PLTL, która nie jest prawdziwa w modelu, w którym czas może się rozgałęziać.

Zadanie 18.6. Wzorując się na konstrukcji PLTL zdefiniuj składnię i semantykę FOLTL (*first order linear temporal logic*), tj. temporalnego rachunku predykatów pierwszego rzędu z czasem liniowym.

Rozdział 19

Programowanie logiczne

Do uzupełnienia:

19.1. Wprowadzenie do Prologu. Dane atomowe, fakty i reguły. Prologowa baza wiedzy. Cel. Sukces, niepowodzenie i nawroty. Konwersacja z interpreterem Prologu
19.2. Dane nieatomowe. Funktory, termy, struktury
19.3. Listy w Prologu
19.4. Rekursja w Prologu, rekursja ogonowa
19.5. Arytmetyka w Prologu
19.6. Prologowe drzewo poszukiwań. Unifikacja celu z głową reguły
19.7. Odcięcie
19.7.1. Negacja jako niepowodzenie. Koncepcja świata zamkniętego
19.8. Metodologia programowania w Prologu
19.8.1. Akumulator
19.8.2. Struktury otwarte
19.9. Zadania

Literatura zastępcza: [30, 159]

Rozdział 20

Lingwistyka porównawcza języków programowania

Do uzupełnienia:

20.1. Sortowanie
20.2. Sito Eratostenesa

Rozdział 21

Zastosowania teorii języków programowania

Do uzupełnienia:

21.1. AnnoDomini: Problem Roku 2000 w COBOL-u i jego rozwiązanie za pomocą systemu typów [49]
21.2. Proof-carrying code [125]

Literatura

[1] Martín Abadi, Luca Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.

[2] Harold Abelson, Gerald J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

[3] Samson Abramsky, Chris Hankin, (eds.) *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[4] P. B. Andrews, *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[5] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley, 1986.

[6] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, *Projektowanie i analiza algorytmów komputerowych*, PWN, 1983.

[7] Saud Alagić, Michael A. Arbib, *Projektowanie programów poprawnych i dobrze zbudowanych*, WNT, 1982.

[8] Lloyd Allison, *A Practical Introduction to Denotational Semantics*, Cambridge University Press, 1996.

[9] Andrew W. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.

[10] Andrew W. Appel, A Critique of Standard ML, *J. Functional Programming* **3** (4), 1993, str. 391-430.

[11] Andrew W. Appel, *Modern Compiler Implementation in ML*, Cambridge University Press, 1998.

[12] Doris Appleby, Julius Vandekopple, *Programming Languages: Paradigm and Practice*, McGraw-Hill, 1997 (wyd. 2).

[13] Robert Laurence Baber, *O programowaniu inaczej*, WNT, 1989.

[14] Rolland C. Backhouse, *Syntax of Programming Languages: Theory and Practice*, Prentice-Hall, 1979.

[15] John Backus, Can programming be liberated from von Neumann style? A functional style and its algebra of programs, *Comm. ACM*, **21**, 1978, str. 613–641.

[16] Henri E. Bal, Dick Grune, *Programming Language Essentials*, Addison-Wesley, 1994.

[17] L. Banachowski, A. Kreczmar, *Elementy analizy algorytmów*, WNT, 1982 (wyd. 1), 1989 (wyd. 2).

[18] M. Ben-Ari, *Podstawy programowania współbieżnego*, WNT, 1989.

[19] M. Ben-Ari, *Podstawy programowania współbieżnego i rozproszonego*, WNT, 1996.

[20] Jon Bentley, *Perełki programowania*, WNT 1992.

[21] Thomas J. Bergin, Richard G. Gibson, *History of Programming Languages*, Volume 2, Addison-Wesley, 1996.

[22] Richard Bird, Philip Wadler, *Introduction to Functional Programming*, Prentice-Hall, 1988.

[23] Richard Bosworth, *A Practical Course in Functional Programming Using Standard ML*, McGraw-Hill, 1995.

[24] Corrado Böhm, Giuseppe Jacopini, Flow diagrams, Turing machines, and languages with only two formation rules, *Communications of the ACM*, **9** (5), 1966, str. 366–371.

[25] Grady Booch, *Object Oriented Design*, Benjamin Cummings, 1991.

[26] Per Brinch Hansen, *Podstawy systemów operacyjnych*, WNT, 1979.

[27] William H. Burge, *Recursive Programming Techniques*, Addison-Wesley, 1975.

[28] Giuseppe Castagna, *Object-Oriented Programming. A Unified Foundation*, Birkhäuser, 1997.

[29] Ruknet Cezzar, *A Guide to Programming Languages: Overview and Comparison*, Artech House, 1995.

[30] William F. Clocksin, Christopher S. Mellish, *Programming in Prolog*, Springer-Verlag, 1987.

[31] Chris Clack, Colin Myers, and Ellen Poon, *Programming with Standard ML*, Prentice-Hall, 1993.

[32] R. G. Clarke, L. B. Wilson, *Comparative Programming Languages*, Addison-Wesley, 1988.

[33] Peter Coad, Edward Yourdon, *Object-oriented Analysis*, Prentice-Hall, 1991.

[34] John L. Connell, Linda Shafer, *Structured Rapid Prototyping*, Prentice-Hall, 1989.

[35] Michael A. Covington, Donald Nute, Andre Vellino, *Prolog Programming in Depth*, Prentice-Hall, 1997.

[36] Brad J. Cox, *Object-oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.

[37] Ole Johan Dahl, Edsger W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[38] Luis Damas, Robin Milner, Principal type-schemes for functional programs, Proc. 9th POPL, 1982, str. 207–212.

[39] Piotr Dembiński, Jan Małuszyński, *Matematyczne metody definiowania języków programowania*, WNT, 1981.

[40] Pierre Deransart, Martin Jourdan, Bernard Lorho, *Attribute Grammars: Definitions, Systems and Bibliography*, LNCS 323, Springer-Verlag, 1988.

[41] Pierre Deransart, A. Ed-Dbali, L. Cervoni, *Prolog. The Standard. Reference Manual*, Springer-Verlag, 1996.

[42] Herbert L. Dershem, Michael J. Jipping, *Programming Languages: Structures and Models*, PWS Publishing Company, 1995 (wyd. 2).

[43] Edsger W. Dijkstra, Go to statements considered harmful, *Comm. ACM*, **11** (3), 1968, str. 147–148.

[44] Edsger W. Dijkstra, The structure of ”THE” — multiprogramming system, *Comm. ACM*, **11** (5), 1968, str. 345–346.

[45] Edsger W. Dijkstra, *Umiejętność programowania*, WNT, 1978 (wyd. 1), 1985 (wyd. 2).

[46] R. Kent Dybvig, *Scheme*, WNT, 1987.

[47] E. Allen Emerson, Temporal and modal logic, *Handbook of Theoretical Computer Science*, Elsevier, 1990.

[48] H. Ehrig, B. Mahr, *Fundamentals of algebraic specifications 1. Equations and Initial Semantics*, Springer-Verlag, 1985.

[49] Peter Harry Eidorf, Fritz Heinglein, Christian Mossin, Henning Niss, Morten Heine Sørensen, Mads Tofte, AnnoDomini: From Type Theory to Year 2000 Conversion Tool, *Conf. Record 26th Symp. Principles of Programming Languages*, POPL’99, ACM Press, 1999, str. 1–14.

[50] Matthias Felleisen, Daniel P. Friedman, *The Little MLer*, MIT Press, 1997.

[51] Antony Field, Peter G. Harrison, *Functional Programming*, Addison-Wesley, 1988.

[52] Raphael A. Finkel, *Advanced Programming Language Design*, Addison-Wesley, 1996.

[53] Charles N. Fischer, jr., Richard J. LeBlanc, *Crafting a Compiler*, Benjamin Cummings, 1988.

[54] J. M. Foster, *Przetwarzanie struktur listowych*, PWN, 1976.

[55] J. M. Foster, *Automatyczna analiza składniowa*, PWN, 1976.

[56] L. W. Friedman, *Comparative Programming Languages*, Prentice-Hall, 1991.

[57] Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes, *Essentials of Programming Languages*, MIT Press, 1992 (wyd. 3).

[58] D. Gelertner, S. Jagannathan, *Programming Linguistics*, MIT Press, 1990.

[59] C. Ghezzi, M. Jazayeri, *Programming Language Concepts*, Wiley, 1982.

[60] Stephen Gilmore, *Applicative Programming and Specification*, manuscript.

[61] Stephen Gilmore, *Programming in Standard ML '97: A Tutorial Introduction*, technical report ECS-LFCS-97-364, LFCS, Edinburgh University.

[62] Joseph Goguen, Grant Malcolm, *Algebraic Semantics of Imperative Programs*, MIT Press, 1996.

[63] Michael J. C. Gordon, *Denotacyjny opis języków programowania*, WNT, 1983.

[64] Michael J. C. Gordon, *Programming Language Theory and Its Implementation*, Prentice-Hall, 1988.

[65] David Gries, *Konstrukcja translatorów dla maszyn liczących*, PWN, 1984.

[66] Carl A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1993.

[67] A. Nico Haberman, Dewayne E. Perry, *Ada dla zaawansowanych*, WNT, 1989.

[68] Michael Hansen, Hans Rischel, *Introduction to Programming Using SML*, Addison-Wesley, 1999.

[69] Robert Harper, *Introduction to Standard ML*, technical report ECS-LFCS-86-14, LFCS, Edinburgh University.

[70] Matthew Hennessy, *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*, Wiley, 1990.

[71] C. A. R. Hoare, Niklaus Wirth, An Axiomatic Definition of the Programming Language PASCAL, *Acta Informatica*, **2** (4), 1973, str. 335–355.

[72] C. A. R. Hoare, Monitors: an operating system concept, *Comm. ACM*, **17** (10), 1974, str. 549–557.

[73] John E. Hopcroft, Jeffrey D. Ullman, *Wprowadzenie do teorii automatów, języków i obliczeń*, PWN, 1994.

[74] F. R. A. Hopgood, *Metody kompilacji*, PWN, 1982.

[75] Ellis Horowitz, *Fundamentals of programming languages*, Springer-Verlag, 1984.

[76] Zbigniew Huzar, *Wstęp do programowania współbieżnego*, skrypt Politechniki Wrocławskiej, 1985.

[77] Wacław Iszkowski, Marek Maniecki, *Programowanie współbieżne*, WNT, 1982.

[78] Kathleen Jensen, Niklaus Wirth, *PASCAL: user manual and report*, Springer-Verlag, 1975.

[79] Cliff B. Jones, *Konstrukcja oprogramowania metodą systematyczną*, WNT, 1984.

[80] Samuel N. Kamin, *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[81] Brian W. Kernighan, P. J. Plauger, *Narzędzia programistyczne w Pascalu*, WNT, 1989.

[82] Brian W. Kernighan, Denis M. Ritchie, *Język C*, WNT, 1988.

[83] Brian W. Kernighan, Denis M. Ritchie, *Język ANSI C*, WNT, 1994.

[84] Donald E. Knuth, *Literate Programming*, Center for Study of Language and Information, Leland Stanford Junior University, 1992.

[85] Peter M. Kogge, *The Architecture of Symbolic Computers*, McGraw Hill, 1991.

[86] S. Rao Kosaraju, An analysis of structured programs, *J. of Comp. and System Sci.*, **9** (3), 1994, str. 232–255.

[87] P. J. Landin, A correspondence between Algol 60 and Church's Lambda notation, *Comm. ACM*, **8**, 1965, str. 158–16.

[88] John Launchbury, Erik Meijer, Tim Sheard (eds.), *Advanced Functional Programming*, Second International School, Olympia WA, USA, August 26–30, 1996, Tutorial Text, LNCS 1129, Springer-Verlag, 1996.

[89] Henry F. Ledgard, *Mała księga programowania obiektowego*, WNT, 1998.

[90] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.

[91] Barbara Liskov, J. Guttag, *Abstraction and Specification in Software Development*, MIT Press, 1986.

[92] Bruce MacLennan, *Functional Programming: Practice and Theory*, Addison-Wesley, 1989.

[93] Bruce MacLennan, *Principles of Programming Languages*, Oxford University Press, 1999 (wyd. 3).

[94] Mark Lorenz, *Object-oriented Software Development: A Practical Guide*, Prentice-Hall, 1991.

[95] Kenneth C. Louden, *Programming Languages: Principles and Practice*, Brooks/Cole, 1993.

[96] Kenneth C. Louden, *Compiler Construction: Principles and Practice*, Brooks/Cole, 1997.

[97] John Lyons, *Wstęp do językoznawstwa*, PWN, 1976.

[98] Jan Małuszyński, Krzysztof Piasecki, *Algol 68*, WNT, 1980.

[99] V.S. Manis, J.J. Little, *The Schematics of Computation*, Prentice Hall, 1995.

[100] Zohar Manna, R. Waldinger, *The Logical Basis for Computer Programming*, Addison-Wesley, 1985 (t. 1), 1990 (t. 2).

[101] Michael Marcotty, Henry F. Ledgard, *W kręgu jezyków programowania*, WNT, 1991.

[102] Jacek Martinek, *LISP*, WNT, 1980.

[103] Paul de Mast, Jan-Marten Jansen, Dick Bruin, Jeroen Fokker, Pieter Koopman, Sjaak Smetsers, Marko van Eekelen, Rinus Plasmeijer, *Functional Programming in Clean*, to appear.

[104] Lawrence J. Mazlack, *Structured Problem Solving with PASCAL*, Holt, Rinehart & Winston, 1983.

[105] Thomas J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering*, **2** (4), 1976, str. 308–320.

[106] John McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, *Comm. ACM*, **3**, April 1960, str. 184–195.

[107] John McCarthy, Michael Levin, *et al.*, *LISP 1.5 Programmer's Manual*, MIT, 1966.

[108] John McCarthy, History of LISP, *Proc. ACM Conf. on the History of Programming Languages*, Los Angeles, 1977, przedruk w Richard L. Wexelblat (ed.), *History of programming languages*, Academic Press, 1981.

[109] LISP — Notes on its past and future — 1980, *Proc. 1980 ACM Conf. on LISP and Functional Programming*, ACM Press, 1980.

[110] Bertrand Meyer, *Introduction to the Theory of Programming Languages*, Prentice-Hall, 1991.

[111] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[112] Greg Michaelson, *Elementary Standard ML*, UCL Press, 1995.

[113] George A. Miller, The magical number seven, plus or minus two: some limits on our capacity for processing information, *The Psychological Review*, **63**, March 1956, str. 81–97.

[114] Robert Milne, Christopher Strachey, *A Theory of Programming Language Semantics*, Chapman and Hall, 1976.

[115] Robin Milner, A theory of type polymorphism in programming, *J. Comp. System Sci.*, **17**, 1978, str. 348–375.

[116] Robin Milner, Mads Tofte, Robert Harper, *The Definition of Standard ML*, MIT Press, 1990.

[117] Robin Milner, Mads Tofte, *Commentary on Standard ML*, MIT Press, 1991.

[118] Robin Milner, Mads Tofte, Robert Harper, David MacQueen, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[119] John C. Mitchell, *Foundations for Programming Languages*, MIT Press, 1996.

[120] A. Mycroft, Polymorphic type schemes and recursive definitions, *Proc. Int'l Symp. on Programming*, Toulouse, 1984, str. 217–239.

[121] C. Morgan, *Programming from Specifications*, Prentice-Hall, 1990.

[122] Mark Mullin, *Rapid Prototyping for Object-oriented Systems*, Addison-Wesley, 1990.

[123] Glenford J. Myers, *Projektowanie niezawodnego oprogramowania*, WNT, 1980.

[124] Peter Naur, Revised report on the algorithmic language Algol 60, *Comm. ACM*, **3** (5) 1960, str. 299–314, także w *Numerische Mathematik*, **2**, 1960, str. 106–136, przedruk w [130] t. 1, str. 19–49, polskie tłumaczenie w [134], str. 213–253.

[125] George C. Necula, Proof-Carrying Code, *Conf. Record 24th Symp. Principles of Programming Languages*, POPL'97, ACM Press, 1997, str. 106–119.

[126] G. Nelson, *Systems Programming with Modula 3*, Prentice-Hall, 1991.

[127] Flemming Nielson, *ML With Concurrency*, Springer-Verlag, 1997.

[128] *Norma ISO 7185:1990 Pascal*

[129] *Norma ISO 10206:1990 Extended Pascal*

[130] Peter W. O'Hearn, Robert D. Tennent (eds.), *Algol-like Languages*, tom 1 i 2, Birkhäuser, 1997.

[131] Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998.

[132] Richard A. O'Keefe, *The Craft of Prolog*, MIT Press, 1990.

[133] D. Parnas, On criteria to be used in decomposing systems into modules, *Comm. of the ACM*, April 1972.

[134] Stefan Paszkowski, *Język Algol 60*, PWN, 1965–1978 (wyd. 1–9).

[135] Lawrence C. Paulson, *ML for the Working Programmer*, Cambridge University Press, 1991 (wyd. 1), 1996 (wyd. 2 zmienione).

[136] John Peterson, Kevin Hammond, (eds.) *Report on the Programming Language Haskell: A Non-strict, Purely Functional Language*, v. 1.3, May 1, 1996.

[137] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[138] R. H. Perrot, *Parallel Programming*, Addison Wesley, 1987.

[139] Rinus Plasmeijer, Marko van Eekelen, *The Concurrent Clean Language Report*, v. 1.3, High Level Software Tools B.V. and University of Nijmegen, 1998.

[140] Andrzej K. Plewicki, *Język programowania APL/360*, PWN, 1977.

[141] Terrence W. Pratt, Marvin Zelkowitz, *Programming Languages: Design and Implementation*, Prentice-Hall, 2000 (wyd. 4).

[142] Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, 1981 (wyd. 1), 2001 (wyd. 5).

[143] Ian C. Pyle, *Ada*, WNT, 1986.

[144] Chris Reade, *Elements of Functional Programming*, Addison-Wesley, 1989.

[145] Steve Reeves, Michael Clarke, *Logic for Computer Science*, Addison-Wesley, 1990.

[146] John H. Reppy, *Concurrent Programming in ML*, Cambridge University Press, 1999.

[147] John C. Reynolds, *Theories of Programming Languages*, Cambridge University Press, 1998.

[148] David A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, 1986.

[149] David A. Schmidt, *The Structure of Typed Programming Languages*, MIT Press, 1994.

[150] Michael L. Scott, *Programming Language Pragmatics*, Morgan-Kaufmann, 1999.

[151] Robert W. Sebesta, *Concepts of Programming Languages*, Addison-Wesley, 1999 (wyd. 4).

[152] Ravi Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1989 (wyd. 1), 1996 (wyd. 2).

[153] Ken Slonneger, Barry L. Kurtz, *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*, Addison-Wesley, 1995.

[154] Stefan Sokołowski, *Applicative High Order Programming: the Standard ML Perspective*, Chapman & Hall Computing, 1991.

[155] J. M. Spivey, *Understanding Z*, Cambridge University Press, 1988.

[156] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, 1992.

[157] Ryan Stansifer, *ML Primer*, Prentice-Hall, 1992.

[158] Ryan Stansifer, *The Study of Programming Languages*, Prentice-Hall, 1995.

[159] Leon Sterling, Ehud Shapiro, *The Art of Prolog*, MIT Press, 1994 (wyd. 2).

[160] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, 1997.

[161] Herbert Stoyan, *Early LISP History (1956-1959)*, University of Erlangen-Nürnberg, manuscript

[162] Christopher Strachey, Fundamental concepts in programming languages, *Higher-order and Symbolic Computation*, **13**, 2000, str. 11–49.

[163] Christopher Strachey, Continuations: a mathematical semantics for handling full jumps, *Higher-Order and Symbolic Computation*, **13**, 2000, str. 135–152.

[164] Bjarne Stroustrup, *Język C++*, WNT, 1994.

[165] Bjarne Stroustrup, *Projektowanie i rozwój języka C++*, WNT, 1996.

[166] Dennie Van Tassel, *Praktyka programowania*, WNT, 1982.

[167] Robert D. Tennent, *Principles of Programming Languages*, Prentice-Hall, 1981.

[168] Robert D. Tennent, *Semantics of Programming Languages*, Prentice-Hall, 1991.

[169] P. D. Terry, *Programming Language Translation: A Practical Approach*, Addison-Wesley, 1987.

[170] Ken Thompson, Reflections on Trusting Trust, *Comm. ACM*, **27** (8), 1984, str. 761–763.

[171] Simon Thompson, *Type Theory and Functional Programming*, Addison-Wesley, 1991.

[172] Simon Thompson, *The Craft of Functional Programming*, Addison-Wesley, 1996.

[173] Jerzy Tiuryn, *Wstęp do teorii mnogości i logiki*, Wydział Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego, 1997.

[174] Mads Tofte, *Four lectures on Standard ML*, technical report ECS-LFCS-89-73, LFCS, Edinburgh University, przedruk w [88], str. 208–238.

[175] Mads Tofte, *Compiler Generators: What They Can Do, What They Might Do, and What They Will Probably Never Do*, Springer-Verlag, 1990.

[176] Mads Tofte, *Essentials of Standard ML Modules*, manuscript.

[177] Mads Tofte, *Tips for Computer Scientists On Standard ML*, manuscript.

[178] Władysław M. Turski, *Metodologia programowania*, WNT, 1978 (wyd. 1), 1982 (wyd. 2).

[179] Jeffrey D. Ullman, *Elements of ML programming*, Prentice Hall, 1994 (wyd. 1), 1998 (wyd. 2).

[180] William M. Waite, Gerhard Goos, *Konstrukcja kompilatorów*, WNT, 1989.

[181] David A. Watt, *Programming Languages: Concepts and Paradigms*, Prentice-Hall, 1990.

[182] David A. Watt, *Programming Language Processors*, Prentice-Hall, 1993.

[183] David A. Watt, *Programming Languages: Syntax and Semantics*, Prentice-Hall, 1997.

[184] Peter Wegener, *Języki programowania, struktury informacji i organizacja maszyny cyfrowej*, PWN, 1979.

[185] Zbigniew Weiss, Tadeusz Gruźlewski, *Programowanie współbieżne i rozproszone*, WNT, 1993.

[186] Ann L. Winblad, Samuel D. Edwards, *Object-oriented Software*, Addison-Wesley, 1990.

[187] Glynn Winskel, *The Formal Semantics of Programming Languages. An Introduction*, MIT Press, 1993.

[188] Niklaus Wirth, *Program development by stepwise refinement*, *Comm. ACM*, **14** (4), 1971, str. 221–227.

[189] Niklaus Wirth, *Wstęp do programowania systematycznego*, WNT, 1978 (wyd. 1), 1987 (wyd. 2).

[190] Niklaus Wirth, *Algorytmy + struktury danych = programy*, WNT, 1980 (wyd. 1), 1989 (wyd. 2), 1999 (wyd. 3), 2000 (wyd. 4).

[191] Niklaus Wirth, *Modula 2*, WNT, 1991.

[192] Edward Yourdon, L. Constantine, *Structured Design*, Yourdon Press, 1978.

[193] Edward Yourdon, *Object-oriented Systems Design: An Integrated Approach*, Prentice-Hall, 1994.

Skorowidz