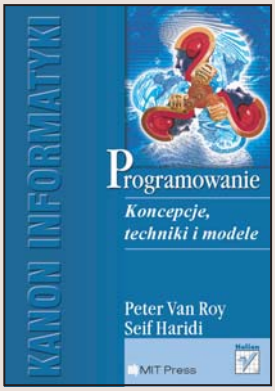
Programowanie. Koncepcje, techniki i modele



Autorzy: Peter Van Roy, Seif Haridi
Tłumaczenie: Bartłomiej Garbacz (wstęp, rozdz. 1–4, 9), Andrzej Grażyński (rozdz. 10–13, dod. A–D), Paweł Koronkiewicz (rozdz. 5–8)
ISBN: 83-7361-979-8
Tytuł oryginału: Concepts, Techniques, and Models of Computer Programming
Format: B5, stron: 860

Poznanie istoty programowania komputerów można zacząć od analizy języków programowania, ich struktur, typów danych i instrukcji. Jednak mnogość języków, różnice pomiędzy nimi i możliwość wykorzystania ich do różnych zadań sprawiają, że przeprowadzenie takiej analizy będzie niezwykle czasochłonne, a jednocześnie nie będzie gwarantowało poznania wszystkich koncepcji i paradygmatów programowania. Naukę koncepcji programowania najlepiej rozpocząć od poznania modelowych struktur realizowanych za pomocą modeli obliczeniowych – konstrukcji definiujących sposób realizacji obliczeń, nie powołujących się na konkretny język.

Książka „Programowanie. Koncepcje, techniki i modele" prezentuje programowanie jako zbiór takich właśnie modeli. Opisuje je w postaci kodów stworzonych w prostym języku podstawowym przeznaczonym dla abstrakcyjnego komputera. W książce przedstawiono zarówno modele ogólne — programowanie deklaratywne, współbieżność deklaratywną, współbieżność przesyłania komunikatów, stan jawny, programowanie zorientowane obiektowo, współbieżność stanu dzielonego oraz programowanie relacyjne — jak i modele specjalizowane, takie jak programowanie graficznych interfejsów użytkownika, programowanie rozproszone oraz programowanie z ograniczeniami. Publikacja zawiera wiele fragmentów programów i ćwiczeń. Można je uruchomić w ramach systemu Mozart Programming System — pakietu programistycznego rozprowadzanego na licencji open source.

- Podstawowe założenia problematyki programowania
- Notacja Backusa-Naura
- Gramatyki kontekstowe i bezkontekstowe
- Zasada działania maszyny abstrakcyjnej
- Typy danych, instrukcje i funkcje
- Drzewa i analiza składniowa
- Metodologie projektowania programów
- Programowanie współbieżne
- Zasady projektowanie i programowanie obiektowego

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

3.

4.

5.

6.

7.

8.

9.

II

10.

11.

2
Deklaratywny model obliczeniowy

Bytów nie należy mnożyć bez konieczności.

— brzytwa Ockhama,
zasada sformułowana przez Williama Ockhama (1285? – 1347/49)

Na programowanie składają się trzy elementy:

- Po pierwsze, model obliczeniowy, który jest formalnym systemem definiującym język, oraz sposoby wykonywania zdań języka (np. wyrażeń i instrukcji) przez maszynę abstrakcyjną. W niniejszej książce jesteśmy zainteresowani modelami obliczeniowymi, które są przydatne i intuicyjnie pojmowane przez programistę. Stanie się to bardziej zrozumiałe, kiedy w dalszej części bieżącego rozdziału zostanie przedstawiony pierwszy z takich modeli.
- Po drugie, zestaw technik programistycznych oraz zasad projektowych wykorzystywanych w celu pisania programów w języku danego modelu obliczeniowego. Niekiedy będziemy określać je mianem modelu programistycznego. Model programistyczny zawsze jest tworzony na podbudowie modelu obliczeniowego.
- Po trzecie, zestaw technik wnioskowania pozwalających analizować programy w celu zwiększenia poziomu zaufania co do poprawności oraz określania wydajności ich działania.

Powyższa definicja modelu obliczeniowego jest bardzo ogólna. Nie wszystkie modele obliczeniowe zdefiniowane w ten sposób będą przydatne dla programistów. Pojawia się zatem pytanie, co można określić jako rozsądny model obliczeniowy. Kierując się intuicją, powiemy, że rozsądny model obliczeniowy to taki, który może być użyty w celu rozwiązywania wielu problemów, który oferuje proste i skuteczne techniki wnioskowania oraz który może być w wydajny sposób zaimplementowany. W dalszej części książki udzielimy bardziej wyczerpującej odpowiedzi na to pytanie. Pierwszym i najprostszym modelem obliczeniowym, jaki omówimy, będzie programowanie deklaratywne. Na razie zdefiniujemy je jako obliczanie funkcji na częściowych strukturach danych. Określa się to niekiedy programowaniem bezstanowym, w przeciwieństwie do programowania stanowego (nazywanego również programowaniem imperatywnym), które zostanie omówione w rozdziale 6.

Model deklaratywny prezentowany w niniejszym rozdziale stanowi jeden z najbardziej podstawowych modeli obliczeniowych. Uwzględnia on kardynalne idee dwóch głównych paradygmatów deklaratywnych, a ściśle programowania funkcyjnego i logicznego. Uwzględnia programowanie z użyciem funkcji operujących na pełnych wartościach, tak jak w językach Scheme i Standard ML. Uwzględnia również deterministyczne programowanie logiczne, tak jak w języku Prolog, w przypadku,

gdy nie używa się operacji wyszukiwania. Wreszcie można uzupełnić go o współbieżność bez utraty jego przydatnych właściwości (patrz rozdział 4.).

Programowanie deklaratywne to duży obszar wiedzy — uwzględnia ono większość idei bardziej ekspresywnych modeli obliczeniowych przynajmniej w podstawowej formie. Stąd też omówimy go w dwóch rozdziałach. W bieżącym rozdziale zdefiniujemy model obliczeniowy oraz bazujący na nim praktyczny język. Następny rozdział będzie prezentował techniki programistyczne tego języka. Kolejne rozdziały będą stanowiły uzupełnienie modelu podstawowego o wiele pojęć. Najważniejsze z nich to obsługa wyjątków, współbieżność, komponenty (w kontekście programowania jako takiego), uprawnienia (w kontekście hermetyzacji i bezpieczeństwa) oraz stan (wprowadzający obiekty i klasy). W kontekście współbieżności omówimy przepływ danych, wykonywanie leniwe, przekazywanie komunikatów, obiekty aktywne, monitory oraz transakcje. Będzie tu również mowa o projektowaniu interfejsu użytkownika, rozproszeniu (z uwzględnieniem odporności na błędy) oraz ograniczeniach (w tym wyszukiwaniu).

Struktura rozdziału

Niniejszy rozdział składa się z ośmiu podrozdziałów:

- W podrozdziale 2.1 wyjaśniono sposób definiowania składni i semantyki praktycznych języków programowania. Składnia zostanie zdefiniowana za pomocą gramatyki bezkontekstowej rozszerzonej o ograniczenia języka. Semantyka zostanie zdefiniowana dwuetapowo: poprzez przetłumaczenie praktycznego języka na prosty język modelowy (ang. *kernel language*), a następnie określenie semantyki języka modelowego. Techniki te będą wykorzystywane w całej książce. W bieżącym rozdziale wykorzystujemy je do zdefiniowania deklaratywnego modelu obliczeniowego.
- Kolejne trzy podrozdziały definiują składnię i semantykę modelu deklaratywnego:
  - W podrozdziale 2.2 omówiono struktury danych: obszar jednokrotnego przypisania i jego zawartość, wartości częściowe oraz zmienne przepływu danych.
  - W podrozdziale 2.3 zdefiniowano składnię języka modelowego.
  - W podrozdziale 2.4 zdefiniowano semantykę języka modelowego w kontekście prostej maszyny abstrakcyjnej. Semantyka została tak zaprojektowana, aby była intuicyjna i pozwalała na proste wnioskowanie na temat poprawności i złożoności.
- W podrozdziale 2.5 została wykorzystana maszyna abstrakcyjna w celu zbadania zachowania obliczeń pod względem wykorzystania pamięci. Zostanie tu omówiona optymalizacja ostatniego wywołania oraz pojęcie cyklu życia pamięci.
- W podrozdziale 2.6 zdefiniowano praktyczny język programowania na podbudowie języka modelowego.
- W podrozdziale 2.7 model deklaratywny rozszerzono o obsługę wyjątków, które pozwalają programom na uwzględnianie nieprzewidzianych i wyjątkowych sytuacji.
- W podrozdziale 2.8 zaprezentowano kilka zaawansowanych zagadnień w celu umożliwienia bardziej zainteresowanym Czytelnikom lepszego zrozumienia modelu.

2.1. Definiowanie praktycznych języków programowania

Języki programowania są znacznie prostsze od języków naturalnych, jednak i tak mogą charakteryzować się zadziwiająco bogatą składnią, zbiorem abstrakcji i bibliotek. Jest to szczególnie widoczne w przypadku języków używanych do rozwiązywania problemów pochodzących z rzeczywistego

świata, które określamy mianem języków praktycznych. Język praktyczny stanowi swego rodzaju skrzynkę z narzędziami doświadczonego mechanika: można tu znaleźć wiele różnych narzędzi służących wielu różnym celom i żadne z tych narzędzi nie znajduje się tam bez powodu.

Niniejszy podrozdział stanowi podstawę dla reszty książki, objaśniając sposób prezentowania składni (gramatyki) oraz semantyki (znaczenia) praktycznych języków programowania. Dzięki tej podstawie będziemy mogli przedstawić pierwszy model obliczeniowy, jakim jest model deklaratywny. Techniki te będą wykorzystywane w dalszej części książki w celu definiowania kolejnych modeli obliczeniowych.

2.1.1. Składnia języka

Składnia języka definiuje, jakie programy są poprawne, tzn. które z nich mogą być poprawnie wykonywane. Na tym etapie nie bierzemy pod uwagę, jakie dokładnie działania wykonują programy. Tu wkroczylibyśmy już w dziedzinę semantyki, którą omówimy w punkcie 2.1.2.

Gramatyki

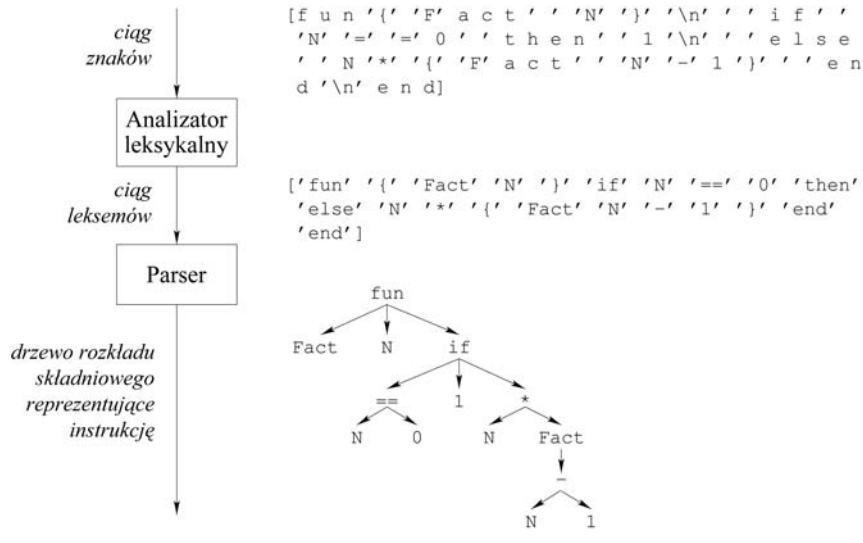
Gramatyka to zestaw reguł definiujących, w jaki sposób można tworzyć „zdania" na podstawie „słów". Gramatyki mogą być używane w przypadku języków naturalnych, takich jak polski lub angielski, jak również w przypadku języków sztucznych, takich jak języki programowania. W przypadku tych ostatnich „zdania" zwykle określa się mianem „instrukcji", zaś „wyrazy" — mianem „leksemów". Tak jak wyrazy składają się z liter, leksemy składają się ze znaków. Daje to nam dwa poziomy struktury:

instrukcja („zdanie")	=	ciąg leksemów („wyrazów")
leksem („wyraz")	=	ciąg znaków („liter")

Gramatyki są przydatne zarówno w zakresie definiowania instrukcji, jak i leksemów. Na rysunku 2.1 przedstawiono przykład pokazujący, w jaki sposób znakowe dane wejściowe są przekształcane w instrukcję. Przykład ten jest definicją funkcji `Fact`:

```
fun {Fact N}
   if N==0 then 1
   else N*{Fact N-1} end
end
```

Danymi wejściowymi jest ciąg znaków, w którym zapis `' '` reprezentuje znak spacji, zaś zapis `'\n'` reprezentuje znak nowego wiersza. Ciąg ten jest najpierw przekształcany na ciąg leksemów, a następnie w drzewo rozkładu składniowego. Składnia obu ciągów z rysunku jest zgodna ze składnią list używaną w całej książce. Choć ciągi są „płaskie", drzewo ukazuje strukturę instrukcji. Program pobierający ciąg znaków i zwracający ciąg leksemów określa się mianem analizatora leksykalnego. Z kolei program pobierający ciąg leksemów i zwracający drzewo analizy składniowej określa się mianem parsera (analizatora składniowego).



RYSUNEK 2.1. Przechodzenie od znaków do instrukcji

Rozszerzona notacja Backusa-Naura

Jedna z najczęściej stosowanych notacji definiowania gramatyk nosi nazwę *rozszerzonej notacji Backusa-Naura* (ang. *Extended Backus-Naur Form*, EBNF). Pochodzi ona od nazwisk jej twórców, Johna Backusa i Petera Naura. Notacja EBNF rozróżnia symbole terminalne i nieterminalne. Symbolem terminalnym jest po prostu leksem. Symbol nieterminalny reprezentuje ciąg leksemów. Definiuje się go za pomocą reguły gramatycznej, która pokazuje, w jaki sposób należy rozwijać ją w leksemy. Przykładowo, poniższa reguła definiuje symbol nieterminalny *⟨digit⟩* (cyfra):

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8| 9

Określa ona, że symbol *⟨digit⟩* reprezentuje jeden z dziesięciu leksemów 0, 1, …, 9. Symbol | odczytuje się jako „lub". Oznacza on możliwość wyboru jednej z alternatyw. Reguły gramatyczne same mogą odwoływać się do symboli nieterminalnych. Przykładowo, możemy zdefiniować symbol nieterminalny *⟨int⟩*, który będzie określał sposób zapisu dodatnich liczb całkowitych:

⟨int⟩ ::= ⟨digit⟩ { ⟨digit⟩ }

Reguła ta określa, że liczba całkowita jest cyfrą, po której występuje dowolna ilość cyfr (być może żadna). Nawiasy klamrowe { i } oznaczają powtórzenie swojej zawartości dowolną liczbę razy, w tym ani razu.

Sposób czytania gramatyk

W celu odczytania gramatyki rozpoczynamy od dowolnego symbolu nieterminalnego, na przykład *⟨int⟩*. Odczytanie odpowiadającej mu reguły gramatycznej od strony lewej do prawej daje ciąg leksemów według następującego schematu:

- Każdy napotkany symbol terminalny jest dodawany do ciągu.
- Dla każdego napotkanego symbolu nieterminalnego odczytujemy jego regułę gramatyczną i zastępujemy taki symbol ciągiem leksemów, na jaki zostanie rozwinięty.
- Za każdym razem, gdy zostanie napotkany wybór (znak |) wybieramy dowolną z alternatyw.

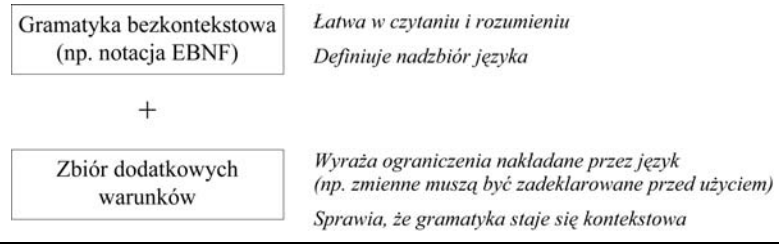
Gramatyka może być używana zarówno w celu sprawdzenia, czy instrukcje są poprawne, jak i w celu ich generowania.

Gramatyki bezkontekstowe i kontekstowe

Każdy dobrze zdefiniowany zbiór instrukcji nazywamy językiem formalnym lub po prostu językiem. Przykładowo, zbiór wszystkich możliwych instrukcji generowanych przez gramatykę i jeden symbol nieterminalny jest językiem. Techniki definiowania gramatyk można sklasyfikować w odniesieniu do ich ekspresywności, tzn. tego, jakie rodzaje języków potrafią generować. Przykładowo, przedstawiona powyżej notacja EBNF definiuje klasę gramatyk określanych mianem gramatyk bezkontekstowych. Nazwa ta wzięła się stąd, że rozwinięcie symbolu nieterminalnego, np. ⟨digit⟩, zawsze daje ten sam wynik bez względu na to, gdzie zostanie wykonane.

W przypadku większości praktycznych języków programowania zazwyczaj nie istnieje jakakolwiek gramatyka bezkontekstowa, która generowałaby wszystkie poprawne programy i żadne inne. Przykładowo, w przypadku wielu języków zmienna musi zostać zadeklarowana, zanim zostanie użyta. Warunku tego nie można wyrazić w ramach gramatyki bezkontekstowej, ponieważ symbol nieterminalny używający zmiennej musi dopuszczać używanie wyłącznie już zadeklarowanych zmiennych. Jest to zależność kontekstowa. Gramatykę zawierającą symbol nieterminalny, którego użycie zależy od kontekstu, w jakim jest używany, określa się mianem gramatyki kontekstowej.

Składnia większości praktycznych języków programowania jest zatem definiowana w dwóch częściach (patrz rysunek 2.2) — jako gramatyka bezkontekstowa uzupełniona o zbiór dodatkowych warunków nakładanych przez język. Gramatyka bezkontekstowa jest przechowywana zamiast pewnej bardziej ekspresywnej notacji, ponieważ jest łatwa w czytaniu i rozumieniu. Charakteryzuje się ona ważną cechą lokalności: symbol nieterminalny można zrozumieć, badając tylko reguły potrzebne do jego zdefiniowania; prawdopodobnie o wiele liczniejsze reguły, które go używają, mogą być zignorowane. Gramatyka bezkontekstowa zostaje poprawiona poprzez nałożenie zbioru dodatkowych warunków, takich jak — w przypadku zmiennych — konieczności deklaracji przed użyciem. Uwzględnienie tych warunków daje gramatykę kontekstową.



RYSUNEK 2.2. Podejście bezkontekstowe do składni języka

Niejednoznaczność

Gramatyki bezkontekstowe mogą być niejednoznaczne, tzn. może istnieć kilka drzew rozkładu składniowego, które odpowiadają danemu ciągowi leksemów. Przykładowo, poniżej przedstawiono prostą gramatykę dla wyrażeń arytmetycznych dodawania i mnożenia:

```
⟨exp⟩ ::= ⟨int⟩ | ⟨exp⟩ ⟨op⟩ ⟨exp⟩
⟨op⟩  ::= + | *
```

Wyrażenie 2*3+4 posiada dwa drzewa rozkładu składniowego w zależności od tego, jak odczytamy dwa wystąpienia symbolu *⟨exp⟩*. Na rysunku 2.3 przedstawiono oba te drzewa. W jednym z nich pierwszym wyrażeniem *⟨exp⟩* jest 2, zaś drugim wyrażeniem *⟨exp⟩* jest 3+4. W drugim drzewie symbolami tymi są, odpowiednio, 2*3 i 4.

RYSUNEK 2.3. Niejednoznaczność w przypadku gramatyki bezkontekstowej

Niejednoznaczność zwykle stanowi niepożądaną właściwość gramatyki, gdyż nie jest wówczas jasne, jaki program został napisany. W przypadku wyrażenia 2*3+4 dwa drzewa rozkładu składniowego dają różne wyniki w razie wyliczenia wyrażeń: pierwsze daje wynik 14 (wynik wykonania obliczenia 2∗(3+4)), zaś drugie daje wynik 10 (wynik wykonania obliczenia (2∗3)+4). Czasem reguły gramatyki można przepisać tak, aby usunąć niejednoznaczność, jednak może to komplikować reguły. Wygodniejszym sposobem jest dodanie dodatkowych warunków. Warunki te ograniczają parser, tak aby było możliwe wygenerowanie tylko jednego drzewa rozkładu składniowego. Mówimy, że usuwają one niejednoznaczność z gramatyki.

W przypadku wyrażeń używających operatorów dwuargumentowych, takich jak przedstawione powyżej wyrażenia arytmetyczne, zwykle stosowanym podejściem jest dodanie dwóch warunków — pierwszeństwa i łączności.

- Pierwszeństwo jest warunkiem nakładanym na wyrażenie zawierające różne operatory, na przykład 2*3+4. Każdemu z operatorów przypisuje się określony poziom pierwszeństwa. Operatory o wysokim pierwszeństwie są umieszczane jak najniżej w drzewie rozkładu składniowego, tzn. jak najdalej od korzenia drzewa. Jeżeli operator * ma wyższy poziom pierwszeństwa niż operator +, to wybrane zostaje drzewo (2*3)+4 zamiast alternatywnego 2*(3+4). Jeżeli operator * znajduje się niżej w drzewie od operatora +, to mówimy, że operator * wiąże silniej od operatora +.
- Łączność jest warunkiem nakładanym na wyrażenie zawierającym ten sam operator, na przykład 2-3-4. W takim przypadku pierwszeństwo nie wystarczy do usunięcia niejednoznaczności, ponieważ wszystkie operatory mają to samo pierwszeństwo. Musimy dokonać wyboru między drzewami (2-3)-4 a 2-(3-4). Łączność określa, czy silniej wiąże operator znajdujący się po lewej czy po prawej stronie. Jeżeli łączność operatora - jest lewostronna, wybrane zostanie drzewo (2-3)-4. Jeżeli natomiast łączność operatora - jest prawostronna, wybrane zostanie drzewo 2-(3-4).

Pierwszeństwo i łączność wystarczą do usunięcia niejednoznaczności związanych ze wszystkimi wyrażeniami definiowanymi przy użyciu operatorów. W dodatku C przedstawiono pierwszeństwo i łączność wszystkich operatorów używanych w książce.

Notacja składniowa używana w książce

W niniejszym rozdziale i pozostałej części książki każdy nowy typ danych i nowa konstrukcja językowa będą wprowadzane razem z niewielkim diagramem składniowym, które będzie pokazywał, jak mają się one do języka jako takiego. Diagram składniowy prezentuje reguły gramatyczne dla prostej bezkontekstowej gramatyki leksemów. Notację tę starannie zaprojektowano w celu spełnienia dwóch podstawowych zasad:

- Wszystkie reguły gramatyczne są niezależne. Żadne później prezentowane informacje nie sprawią, że reguła taka stanie się niepoprawna. Oznacza to, że nigdy nie będą podawane niepoprawne reguły gramatyczne wyłącznie w celu „uproszczenia" prezentacji.
- Na podstawie inspekcji zawsze można wyraźnie stwierdzić, kiedy reguła gramatyczna całościowo definiuje symbol nieterminalny lub kiedy prezentuje tylko częściową definicję. Definicja częściowa zawsze kończy się wielokropkiem . . .

Wszystkie diagramy składniowe używane w książce zebrano w dodatku C. Przedstawiono w nim również składnię leksykalną leksemów w kontekście znaków. Poniżej podano przykład diagramu składniowego z dwiema regułami gramatycznymi, które ilustrują stosowaną notację:

```
⟨statement⟩  ::= skip | ⟨expression⟩ '=' ⟨expression⟩ | ...
⟨expression⟩ ::= ⟨variable⟩ | ⟨int⟩ | ...
```

Powyższe reguły stanowią częściową definicję dwóch symboli nieterminalnych, *⟨statement⟩* (instrukcja) oraz *⟨expression⟩* (wyrażenie). Pierwsza reguła stwierdza, że instrukcją może być słowo kluczowe `skip` lub dwa wyrażenia rozdzielone symbolem równości = albo pewien inny element. Druga reguła stwierdza, że wyrażenie może być zmienną, liczbą całkowitą lub innym elementem. Wybór między różnymi możliwościami w regule gramatycznej oznacza pionowa kreska |. W celu uniknięcia niedomówień związanych ze składnią samej reguły gramatycznej czasem będziemy umieszczać w apostrofach symbol, który dosłownie występuje w tekście. Przykładowo, symbol równości przedstawiono jako `'='`. Słowa kluczowe nie będą ujmowane w apostrofy, gdyż w ich przypadku nie ma mowy o niejednoznaczności.

Poniżej przedstawiono kolejny przykład notacji:

```
⟨statement⟩  ::= if ⟨expression⟩ then ⟨statement⟩
                   { elseif ⟨expression⟩ then ⟨statement⟩ }
                   [ else ⟨statement⟩ ] end | ...
⟨expression⟩ ::= '[' { ⟨expression⟩ }+ ']' | ...
⟨label⟩      ::= unit | true | false | ⟨variable⟩ | ⟨atom⟩
```

Pierwsza reguła definiuje instrukcję `if`. Występuje tu opcjonalna sekwencja klauzul `elseif`, tzn. może wystąpić dowolna ich liczba, w tym zero. Sugerują to nawiasy klamrowe { . . . }. Dalej występuje opcjonalna klauzula `else`, tzn. może ona wystąpić zero lub jeden raz. Mówią o tym nawiasy kwadratowe [. . .]. Druga reguła definiuje składnię list jawnych. Muszą one posiadać co najmniej jeden element, np. `[5 6 7]` jest listą poprawną, ale `[ ]` już nie (należy zwrócić uwagę na znak spacji

oddzielający nawiasy [i]). Sugeruje to zapis {...}+. Trzecia reguła definiuje składnię etykiet rekordów. Stanowi ona definicję pełną, gdyż nie występuje tu znak wielokropka. Istnieje pięć możliwości i nigdy nie zostanie określona większa ich liczba.

2.1.2. Semantyka języka

Semantyka języka definiuje, co program robi w czasie działania. W sytuacji idealnej semantyka powinna być zdefiniowana w ramach prostej matematycznej struktury, która pozwala wnioskować na temat programu (w tym odnośnie do jego poprawności, czasu wykonania oraz użycia pamięci) bez wprowadzania nieistotnych szczegółów. Czy można to osiągnąć w przypadku praktycznego języka bez zbytniego komplikowania semantyki? Używana przez nas technika, którą określamy mianem podejścia opartego na języku modelowym, stanowi odpowiedź twierdzącą na tak postawione pytanie.

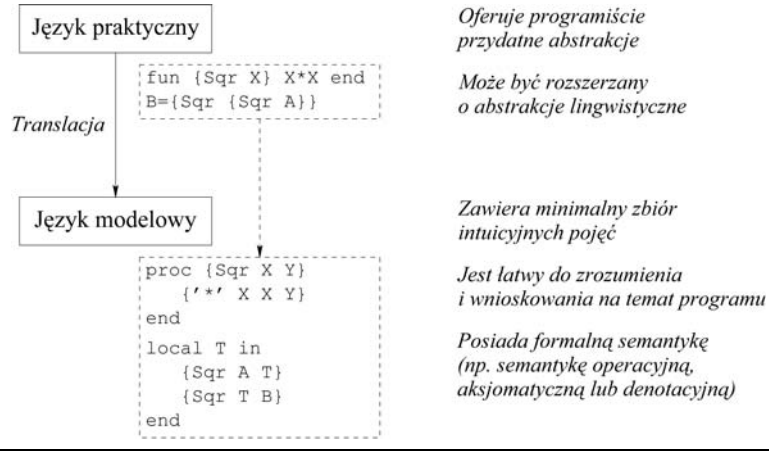
Współczesne języki programowania ewoluowały przez ponad pięć dekad doświadczeń związanych z konstruowaniem rozwiązań programistycznych złożonych problemów pochodzących z rzeczywistego świata[1]. Współczesne programy mogą być bardzo złożone, osiągając miliony wierszy kodu, pisane przez duże zespoły programistów przez wiele lat. Zdaniem autorów języki umożliwiające skalowanie do takiego poziomu złożoności okazały się sukcesem po części dlatego, że modelują pewne najistotniejsze aspekty sposobu konstruowania złożonych programów. W tym sensie języki te nie są tylko dowolnymi konstrukcjami ludzkiego umysłu. Warto byłoby więc dogłębnie je zrozumieć dzięki użyciu metod naukowych, tzn. wyjaśniając ich zachowanie w kontekście prostego, odpowiadającego im modelu. Stanowi to jeden z głównych powodów wykorzystania podejścia opartego na języku modelowym.

Podejście oparte na języku modelowym

W niniejszej książce podejście oparte na języku modelowym jest wykorzystywane w celu definiowania semantyk języków programowania. W przypadku tego podejścia wszystkie konstrukcje języka są definiowane w kontekście translacji na język modelowy. Podejście oparte na języku modelowym uwzględnia dwa elementy (rysunek 2.4.):

- Po pierwsze, definiujemy bardzo prosty język, nazywany językiem modelowym. Język ten powinien ułatwiać wyciąganie wniosków na temat programów oraz powinien odpowiadać wydajności pamięciowej i czasowej implementacji. Język modelowy oraz struktury danych, na których wykonuje swoje działania, wspólnie tworzą *rdzenny model obliczeniowy* (ang. *kernel computation model*).
- Po drugie, definiujemy schemat translacji z pełnego języka programowania na język modelowy. Każda konstrukcja gramatyczna języka pełnego jest tłumaczona na język modelowy. Translacja ta powinna być jak najprostsza. Istnieją dwa rodzaje translacji: abstrakcja lingwistyczna i lukier syntaktyczny. Zostaną one omówione poniżej.

[1] Wartość pięciu dekad jest dość umowna. Jako datę początkową określamy pierwszy działający komputer obsługujący programy składowane — Manchester Mark I. Według dokumentów laboratoryjnych wykonał on swój pierwszy program 21 czerwca 1948 roku [197].



RYSUNEK 2.4. Podejście do semantyki oparte na języku modelowym

Podejście oparte na języku modelowym jest wykorzystywane w całej książce. Każdy model obliczeniowy posiada język modelowy, który bazuje na swoim poprzedniku i uwzględnia jedno nowe pojęcie. Pierwszy język modelowy prezentowany w niniejszym rozdziale nosi nazwę deklaratywnego języka modelowego. W dalszej części książki zostanie zaprezentowanych wiele innych języków modelowych.

Semantyka formalna

Podejście oparte na języku modelowym pozwala na definiowanie semantyki takiego języka w dowolny sposób. Można wyróżnić cztery powszechnie używane podejścia do kwestii semantyki języka:

- Semantyka operacyjna pokazuje, w jaki sposób instrukcja jest wykonywana w kontekście maszyny abstrakcyjnej. Podejście takie zawsze działa dobrze, gdyż wszystkie języki, jakby nie patrzeć, są wykonywane na komputerze.
- Semantyka aksjomatyczna definiuje semantykę instrukcji jako związek między stanem wejściowym (sytuacją przed wykonaniem instrukcji) a stanem wyjściowym (sytuacją po wykonaniu instrukcji). Związek ten jest określany w formie asercji logicznej. Jest to dobry sposób wnioskowania na temat ciągów instrukcji, gdyż asercja wyjściowa każdej z nich stanowi asercję wejściową następnej. Dlatego też schemat ten sprawdza się w przypadku modeli stanowych, gdyż stan jest ciągiem wartości. W podrozdziale 6.6 zostanie przedstawiona semantyka aksjomatyczna modelu stanowego.
- Semantyka denotacyjna definiuje instrukcję jako funkcję dziedziny abstrakcyjnej. Sprawdza się to w przypadku modelu deklaratywnego, ale może być stosowane również w przypadku innych modeli. Sprawy komplikują się, kiedy pojawia się kwestia języków współbieżnych. W punktach 2.8.1 i 4.9.2 zostanie wyjaśnione programowanie funkcyjne, które jest szczególnie bliskie semantyce denotacyjnej.
- Semantyka logiczna definiuje instrukcję jako model teorii logicznej. Sprawdza się to w przypadku modeli obliczeniowych deklaratywnego i relacyjnego, ale jest trudne do zastosowania w przypadku innych. W podrozdziale 9.3 zostanie przedstawiona semantyka logiczna dwóch wyżej wspomnianych modeli obliczeniowych.

Duża część teorii związanej z tymi różnymi semantykami jest interesująca głównie z punktu widzenia matematyków, a nie programistów. Prezentacja tej teorii wykracza poza zakres tematyczny niniejszej książki. Główną semantyką formalną prezentowaną przez nas będzie semantyka operacyjna. Zdefiniujemy ją dla każdego modelu obliczeniowego. Jest ona na tyle szczegółowa, że okazuje się przydatna w zakresie wnioskowania o poprawności i złożoności, a jednocześnie na tyle abstrakcyjna, że pozwala uniknąć niepotrzebnego zamieszania. W rozdziale 13. zebrano wszystkie te semantyki operacyjne w ramach pojedynczego formalizmu o zwięzłej i czytelnej notacji.

W całej książce będą przedstawiane nieformalne semantyki każdej nowej konstrukcji językowej i często będziemy nieformalnie wnioskować na temat działania programów. Takie nieformalne prezentacje zawsze bazują na semantykach operacyjnych.

Abstrakcja lingwistyczna

Zarówno języki programowania, jak i języki naturalne mogą ewoluować, tak aby spełniać stawiane im wymagania. Gdy korzystamy z języka programowania, w pewnym momencie możemy uznać, że konieczne jest rozszerzenie języka, tzn. dodanie nowej konstrukcji lingwistycznej. Przykładowo, model deklaratywny, opisywany w niniejszym rozdziale, nie uwzględnia żadnych konstrukcji pętlowych. W punkcie 3.6.3 zostanie zdefiniowana konstrukcja `for` w celu wyrażania określonych rodzajów pętli, które są przydatne w zakresie pisania programów deklaratywnych. Ta nowa konstrukcja stanowi zarówno abstrakcję, jak i uzupełnienie składni języka — stąd nazywamy ją abstrakcją lingwistyczną. Praktyczny język programowania zawiera wiele abstrakcji lingwistycznych.

Można wyróżnić dwie fazy definiowania abstrakcji lingwistycznej. W pierwszej definiujemy nową konstrukcję gramatyczną. W drugiej definiujemy jej przekształcenie na język modelowy. Język modelowy nie ulega zmianie. Niniejsza książka zawiera wiele przykładów przydatnych abstrakcji lingwistycznych, np. funkcje (`fun`), pętle (`for`), funkcje leniwe (`fun lazy`), klasy (`class`), blokady wielobieżne (`lock`) i inne[2]. Niektóre z nich stanowią element systemu Mozart. Inne można do niego dodać za pomocą narzędzia `gump` [117]. Opis używania tego narzędzia wykracza jednak poza zakres tematyczny niniejszej książki.

Niektóre języki oferują mechanizmy służące do programowania abstrakcji lingwistycznych bezpośrednio w języku. Prostym, a jednocześnie oferującym ogromne możliwości przykładem jest makro języka Lisp. Makro takie przypomina funkcję generującą kod Lispa w czasie wywołania. Częściowo ze względu na prostotę składni Lispa makra te okazały się ogromnym sukcesem, zarówno w samym Lispie, jak i jego następcach. Lisp oferuje wewnętrzną obsługę makr, na przykład cytowanie (zamiana wyrażenia programu na strukturę danych) oraz cytowanie wsteczne (działanie odwrotne — w ramach przytaczanej struktury). Szczegółowe omówienie makr Lispa i związanych z tym pojęć Czytelnik znajdzie w każdej dobrej książce poświęconej temu językowi [72, 200].

Prostym przykładem abstrakcji lingwistycznej jest funkcja, która używa słowa kluczowego `fun`. Wyjaśniono to w punkcie 2.6.2. W rozdziale 1. używaliśmy już funkcji, jednak język modelowy z niniejszego rozdziału zawiera tylko procedury. Są one używane dlatego, że wszystkie argumenty są jawne i może występować kilka danych wyjściowych. Można wyróżnić inne, bardziej przemyślane powody wyboru procedur, co zostanie zrobione w dalszej części rozdziału. Jednak ze względu na fakt, że funkcje są bardzo przydatne, dodajemy je jako abstrakcję lingwistyczną.

Definiujemy składnię zarówno dla definicji, jak i wywołań funkcji oraz translację na definicje i wywołania procedur. Taka translacja pozwala nam odpowiedzieć na wszystkie pytania dotyczące wywołań funkcji. Przykładowo, co dokładnie oznacza zapis `{F1 {F2 X} {F3 Y}}` (zagnieżdżone

[2] Bramki logiczne (`gate`) w przypadku opisu obwodów, skrzynki odbiorcze (`receive`) w przypadku współbieżności z przesyłaniem komunikatów oraz currying i wyobrażenie listy (ang. *list comprehension*) występujące we współczesnych językach funkcyjnych — na przykład w języku Haskell.

wywołania funkcji)? Czy porządek wywołań tych funkcji został zdefiniowany? Jeżeli tak, jaki on jest? Istnieje wiele możliwości. Niektóre języki nie precyzują kolejności ewaluacji argumentów, ale zakłada się wówczas, że argumenty funkcji są ewaluowane przed nią. W przypadku innych języków zakłada się, że argument jest ewaluowany, gdy i jeśli jego wynik jest potrzebny, ale nie wcześniej. Tak więc nawet tak prosta rzecz jak wywołania funkcji zagnieżdżonych nie musi mieć oczywistej semantyki. Translacja precyzyjnie określa, jaka jest semantyka.

Abstrakcje lingwistyczne są przydatne nie tylko w zakresie zwiększania ekspresywności programu. Mogą one również polepszać inne właściwości, takie jak poprawność, bezpieczeństwo oraz wydajność. Ukrywając implementację abstrakcji przed programistą, wsparcie lingwistyczne sprawia, że nie jest możliwe niepoprawne użycie abstrakcji. Kompilator może użyć tych informacji w celu utworzenia wydajniejszego kodu.

Lukier składniowy

Często przydatną rzeczą jest udostępnienie skrótowej notacji dla często występujących idiomów. Taka notacja stanowi element składni języka i jest definiowana za pomocą reguł gramatycznych. Notację tę określa się mianem *lukru składniowego* (ang. *syntactic sugar*). Lukier składniowy stanowi analogię do abstrakcji lingwistycznej o tyle, że jego znaczenie zostaje ściśle zdefiniowane poprzez translację na pełny język. Nie należy jednak mylić go z abstrakcją lingwistyczną — nie oferuje nowej abstrakcji, a tylko redukuje rozmiar programu i zwiększa jego czytelność.

Poniżej zostanie przedstawiony przykład lukru składniowego, który bazuje na instrukcji `local`. Zmienne lokalne zawsze mogą być definiowane przy użyciu instrukcji `local X in ... end`. Kiedy instrukcja ta zostanie zastosowana wewnątrz innej, wygodnie jest posiadać lukier składniowy, który pozwala na pominięcie słów kluczowych `local` i `end`. Zamiast pisać:

```
if N==1 then [1]
else
   local L in
      ...
   end
end
```

możemy użyć zapisu:

```
if N==1 then [1]
else L in
   ...
end
```

który jest zarówno bardziej zwięzły, jak i czytelniejszy od pełnej wersji notacji. Inne przykłady lukru składniowego podano w punkcie 2.6.1.

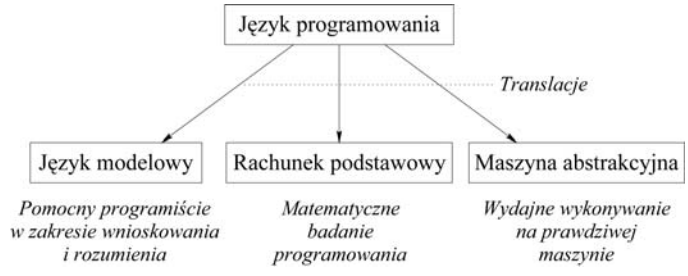
Projektowanie języka

Abstrakcje lingwistyczne stanowią podstawowe narzędzie projektowania języka. Posiadają one naturalne miejsce w cyklu życia abstrakcji. Abstrakcja posiada trzy fazy swojego cyklu życia. Kiedy najpierw jest definiowana, nie posiada żadnego wsparcia lingwistycznego, tzn. w ramach języka nie istnieje żadna konstrukcja składniowa, która ułatwiałaby użycie abstrakcji. Jeżeli w pewnym momencie uzna się, że jest ona podstawowa i bardzo przydatna, możemy zdecydować się dodać

do niej wsparcie lingwistyczne. Wtedy staje się ona abstrakcją lingwistyczną. Jest to faza badania, tzn. nie ma jeszcze żadnej pewności, że ta abstrakcja lingwistyczna stanie się częścią języka. Jeżeli jednak okaże się sukcesem, tzn. upraszcza programy i jest przydatna dla programistów, staje się częścią języka.

Inne podejścia wykorzystujące translację

Podejście oparte na języku modelowym stanowi przykład podejścia do semantyki wykorzystującego translację, tzn. jest ono oparte na translacji z jednego języka na drugi. Na rysunku 2.5 zobrazowano trzy sposoby użycia podejścia translacyjnego w celu definiowania języków programowania:



RYSUNEK 2.5. Podejścia translacyjne do semantyki języka

- Podejście oparte na języku modelowym, używane w niniejszej książce, jest przeznaczone dla programistów. Jego pojęcia są bezpośrednio związane z pojęciami programistycznymi.
- Podejście podstawowe jest przeznaczone dla matematyków. Przykładami są tu: maszyna Turinga, rachunek λ (leżący u podstaw programowania funkcyjnego), logika pierwszego rzędu (leżąca u podstaw programowania logicznego) oraz rachunek π (służący do modelowania współbieżności). Ze względu na fakt, że te rachunki mają w zamierzeniu stanowić przedmiot formalnych badań matematycznych, posiadają jak najmniejszą liczbę elementów.
- Podejście oparte na maszynie abstrakcyjnej jest przeznaczone dla implementatorów. Programy są tłumaczone na kod wyidealizowanej maszyny, którą określa się mianem *maszyny abstrakcyjnej* (ang. *abstract machine*) lub *maszyny wirtualnej* (ang. *virtual machine*)[3]. Stosunkowo prostą rzeczą jest dokonanie translacji kodu wyidealizowanej maszyny na kod maszyny rzeczywistej.

Ze względu na fakt, że skupiamy się na praktycznych technikach programistycznych, w niniejszej książce używamy tylko podejścia opartego na języku modelowym. Pozostałe dwa podejścia wiążą się z problemem polegającym na tym, że każdy zapisany w nich rzeczywisty program jest uwikłany w szczegóły techniczne dotyczące mechanizmów języka. Podejście oparte na języku modelowym pozwala tego uniknąć poprzez odpowiedni dobór pojęć.

[3] Ściśle rzecz ujmując, maszyna wirtualna jest programową emulacją rzeczywistej maszyny działającą w jej ramach i jest niemal tak wydajna jak maszyna rzeczywista. Wydajność tę osiąga się, wykonując większość instrukcji wirtualnych bezpośrednio jako instrukcje rzeczywiste. Koncepcję tę wprowadziła firma IBM na początku lat 60. XX w. w przypadku systemu operacyjnego VM. Ze względu na sukces języka Java, w którego przypadku stosuje się pojęcie „maszyny wirtualnej", współczesne użycie tego pojęcia uwzględnia również sens maszyny abstrakcyjnej.

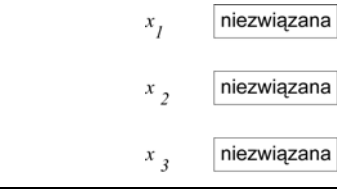
Podejście oparte na interpreterze

Alternatywą dla podejścia translacyjnego jest podejście oparte na wykorzystaniu interpretera. Semantyka języka zostaje zdefiniowana poprzez określenie interpretera języka. Nowe funkcje języka są definiowane poprzez rozszerzanie interpretera. Interpreter jest programem napisanym w języku L_1, który akceptuje programy napisane w innym języku L_2 i wykonuje je. Podejście jest wykorzystywane w książkach Abelsona i Sussmana oraz Sussmana [2]. W ich przypadku interpreter jest metacykliczny (ang. *metacircular*), tzn. języki L_1 i L_2 są tym samym językiem L. Dodanie nowej funkcji języka, np. obsługi współbieżności i wykonywania leniwego, daje nowy język L', który jest implementowany przez rozszerzenie interpretera języka L.

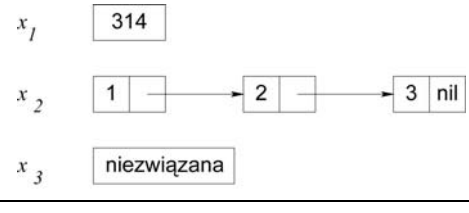
Podejście oparte na interpreterze daje tę korzyść, że ukazuje samodzielną implementację abstrakcji lingwistycznych. W niniejszej książce nie korzystamy z tego podejścia, ponieważ w ogólności nie zapewnia ono zachowania złożoności czasu wykonania programów (liczby potrzebnych operacji w funkcji rozmiaru danych wejściowych). Druga trudność wiążę się z tym, że podstawowe pojęcia znajdują się w interpreterze nawzajem w interakcji, co utrudnia ich zrozumienie. Podejście translacyjne ułatwia zachowanie oddzielenia pojęć.

2.2. Obszar jednokrotnego przypisania

Model deklaratywny wprowadzimy, opisując w pierwszej kolejności jego struktury danych. Model ten wykorzystuje *obszar jednokrotnego przypisania* (ang. *single-assignment store*): zbiór zmiennych, które są początkowo niezwiązane i mogą być związane z tylko jedną wartością. Na rysunku 2.6 przedstawiono obszar z trzema zmiennymi niezwiązanymi x_1, x_2 oraz x_3. Obszar ten możemy zapisać jako $\{x_1, x_2, x_3\}$. Na razie zakładamy, że jako wartości możemy używać liczb całkowitych, list oraz rekordów. Na rysunku 2.7 przedstawiono obszar, w którym zmienna x_1 jest związana z liczbą całkowitą 314, zaś zmienna x_2 — z listą `[1 2 3]`. Zapisujemy to jako $\{x_1 = 314, x_2 = \texttt{[1 2 3]}, x_3\}$.



RYSUNEK 2.6. Obszar jednokrotnego przypisania z trzema zmiennymi niezwiązanymi



RYSUNEK 2.7. Związanie dwóch zmiennych z wartościami

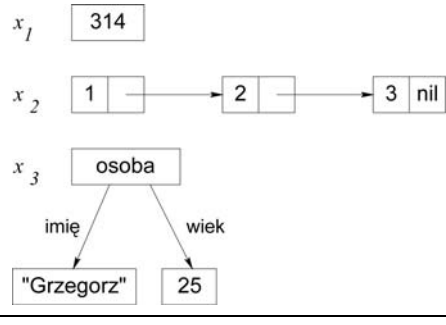
2.2.1. Zmienne deklaratywne

Zmienne znajdujące się w obszarze jednokrotnego przypisania określa się mianem zmiennych deklaratywnych. Pojęcia tego używamy zawsze, kiedy istnieje możliwość pomyłki z innymi zmiennymi. W dalszej części książki zmienne takie będziemy również nazywać zmiennymi przepływu danych ze względu na rolę, jaką odgrywają w wykonywaniu przepływów danych.

Po związaniu zmienna deklaratywna pozostaje związana przez okres obliczeń i jest nieodróżnialna od swojej wartości. Oznacza to, że może ona być używana w obliczeniach tak, jakby była wartością. Wykonanie działania $x+y$ oznacza to samo co wykonanie działania 11+22, jeżeli obszar ma postać $\{x = 11, y = 22\}$.

2.2.2. Obszar wartości

Obszar, w którym wszystkie zmienne są związane z wartościami, określa się mianem *obszaru wartości* (ang. *value store*). Inaczej rzecz ujmując, obszar wartości jest trwałym odwzorowaniem zmiennych na wartości. Wartość jest stałą matematyczną. Przykładowo, liczba całkowita 314 jest wartością. Wartości mogą również być encjami złożonymi, tzn. elementami zawierającymi jedną lub więcej wartości. Na przykład lista `[1 2 3]` oraz rekord `osoba(imię:"Grzegorz" wiek:25)` są wartościami. Na rysunku 2.8 przedstawiono obszar wartości, w którym zmienna x_1 jest związana z liczbą całkowitą 314, zmienna x_2 — z listą `[1 2 3]`, zaś zmienna x_3 — z rekordem `osoba(imię:"Grzegorz" wiek:25)`. Języki funkcyjne, takie jak Standard ML, Haskell i Scheme, wykorzystują tylko obszar wartości, ponieważ obliczają funkcje na wartościach (języki obiektowe, takie jak Smalltalk, C++ i Java, potrzebują obszaru komórek składającego się z komórek, których zawartość można zmieniać).



RYSUNEK 2.8. Obszar wartości: wszystkie zmienne są związane z wartościami

W tym momencie Czytelnik posiadający pewne doświadczenie programistyczne może się zastanawiać, dlaczego wprowadzamy pojęcie obszaru jednokrotnego przypisania, skoro inne języki obywają się bez niego i używają obszaru wartości lub obszaru komórek. Z wielu powodów. Po pierwsze, chcemy wykonywać obliczenia na wartościach częściowych. Przykładowo, procedura może zwracać dane wyjściowe, wiążąc niezwiązaną zmienną przekazaną jako argument. Po drugie, ze względu na współbieżność deklaratywną, która stanowi przedmiot rozdziału 4. Jej użycie jest możliwe dzięki obszarowi jednokrotnego przypisania. Po trzecie, obszar jednokrotnego przypisania jest potrzebny w przypadku programowania relacyjnego (logicznego) oraz programowania z ograniczeniami. Inne powody związane z wydajnością (np. rekurencja końcowa i listy różnicowe) zostaną wyjaśnione w następnym rozdziale.

2.2.3. Tworzenie wartości

Podstawową operacją wykonywaną na obszarze jest związanie zmiennej z nowo utworzoną wartością. Będziemy to zapisywać jako x_i = *wartość*. W zapisie tym x_i bezpośrednio odwołuje się do zmiennej należącej do obszaru (nie jest to nazwa tekstowa zmiennej z programu!), zaś *wartość* odnosi się do wartości, np. 314 lub `[1 2 3]`. Przykładowo, na rysunku 2.7 przedstawiono obszar z rysunku 2.6 po wykonaniu dwóch związań:

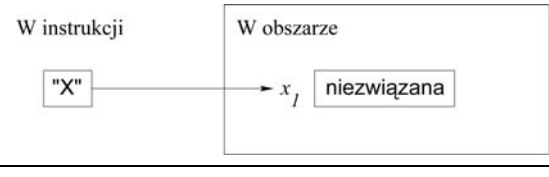
$x_1 = 314$
$x_2 =$ `[1 2 3]`

Operacja jednokrotnego przypisania x_i = *wartość* tworzy *wartość* w obszarze, a następnie wiąże zmienną x_i z tą wartością. Jeżeli zmienna jest już związana, operacja wykonuje sprawdzenie, czy wartości te są zgodne. Jeżeli tak nie jest, zostaje zgłoszony błąd (przy użyciu mechanizmu obsługi wyjątków — patrz podrozdział 2.7).

2.2.4. Identyfikatory zmiennych

Jak dotąd przyglądaliśmy się obszarowi, który zawierał zmienne i wartości, tzn. elementy obszaru, na których można wykonywać obliczenia. Przydatną rzeczą byłoby posiadanie możliwości odwoływania się do elementów obszaru spoza niego. Taką rolę pełnią identyfikatory zmiennych. Identyfikator zmiennej jest nazwą tekstową, która odwołuje się do elementu obszaru spoza niego. Odwzorowanie identyfikatorów zmiennych na elementy obszaru określa się mianem środowiska.

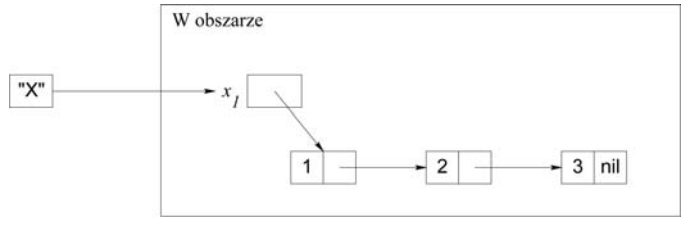
Nazwy zmiennych w kodzie źródłowym programu są w rzeczywistości identyfikatorami zmiennych. Przykładowo, na rysunku 2.9 występuje identyfikator `X`, który odwołuje się do zmiennej obszaru x_1. Odpowiada to środowisku $\{\texttt{X} \rightarrow x_1\}$. Mówiąc o dowolnym identyfikatorze, będziemy używać notacji *⟨x⟩*. Środowisko $\{\langle x \rangle \rightarrow x_1\}$ jest takie samo jak wcześniej, jeżeli *⟨x⟩* reprezentuje `X`. Jak zobaczymy później, identyfikatory zmiennych i odpowiadające im elementy obszaru są dodawane do środowiska za pomocą instrukcji `local` i `declare`.



RYSUNEK 2.9. Identyfikator zmiennej odwołujący się do zmiennej niezwiązanej

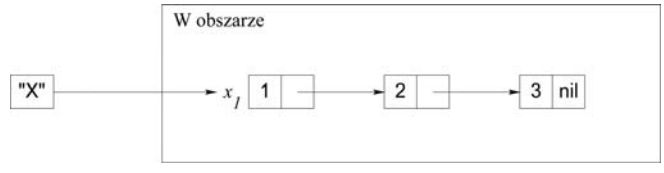
2.2.5. Tworzenie wartości z użyciem identyfikatorów

Po związaniu zmienna jest nieodróżnialna od swojej wartości. Na rysunku 2.10 pokazano, co dzieje się w przypadku, gdy zmienna x_1 z rysunku 2.9 zostanie związana z listą `[1 2 3]`. Przy użyciu identyfikatora zmiennej `X` powiązanie to możemy zapisać jako `X=[1 2 3]`. Jest to tekst, jaki zapisałby programista w celu wyrażenia powiązania. Możemy również użyć notacji *⟨x⟩*`=[1 2 3]`, jeżeli chcemy mieć możliwość mówienia o dowolnym identyfikatorze. W celu zapewnienia poprawności tej notacji w programie zapis *⟨x⟩* musi zostać zastąpiony identyfikatorem.



RYSUNEK 2.10. Identyfikator zmiennej odwołujący się do zmiennej związanej

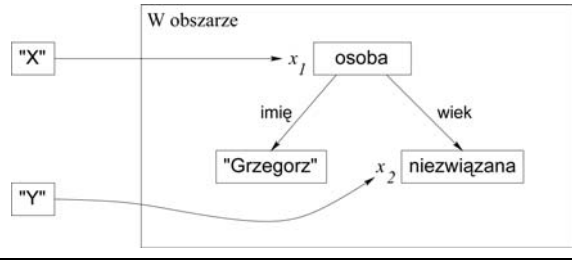
Znak równości „=” odnosi się do operacji wiązania. Po jej zakończeniu identyfikator „`X`” wciąż odwołuje się do zmiennej x_1, która teraz jest związana z wartością `[1 2 3]`. Jest to nieodróżnialne od sytuacji pokazanej na rysunku 2.11, gdzie `X` bezpośrednio odwołuje się do wartości `[1 2 3]`. Śledzenie łączy zmiennych związanych w celu otrzymania wartości określa się mianem dereferencji (wyłuskiwania). Jest to niewidoczne dla programisty.



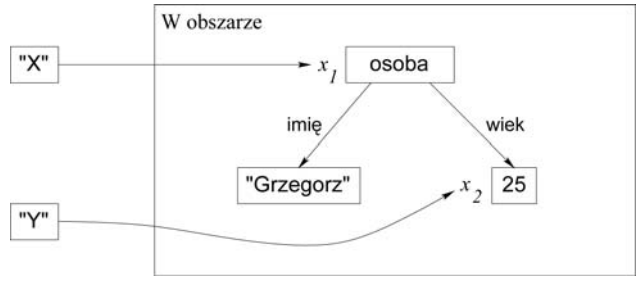
RYSUNEK 2.11. Identyfikator zmiennej odwołujący się do wartości

2.2.6. Wartości częściowe

Wartość częściowa jest strukturą danych, która może zawierać zmienne niezwiązane. Na rysunku 2.12 przedstawiono rekord `osoba(imię:"Grzegorz" wiek:`x_2`)`, do którego odwołuje się identyfikator `X`. Jest to wartość częściowa, ponieważ zawiera zmienną niezwiązaną x_2. Na rysunku 2.13 przedstawiono sytuację po związaniu x_2 z wartością 25 (poprzez operator powiązania `Y=25`). Teraz x_1 jest wartością częściową bez zmiennych niezwiązanych, którą określa się mianem wartości pełnej. Zmienna deklaratywna może zostać związana z kilkoma zmiennymi częściowymi, o ile są one zgodne ze sobą. Mówimy, że zbiór zmiennych częściowych jest zgodny, jeżeli znajdujące się w nim zmienne niezwiązane mogą zostać związane w taki sposób, że wszystkie będą równe. Przykładowo, `osoba(wiek:25)` i `osoba(wiek:`x`)` są zgodne (ponieważ zmienna x może zostać związana z wartością 25), jednak `osoba(wiek:25)` i `osoba(wiek:26)` nie są zgodne.



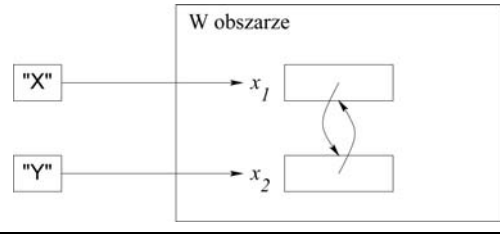
RYSUNEK 2.12. Wartość częściowa



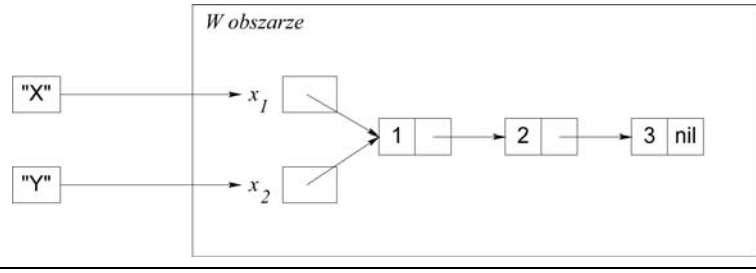
RYSUNEK 2.13. Wartość częściowa bez zmiennych niezwiązanych, czyli wartość pełna

2.2.7. Wiązanie typu zmienna-zmienna

Zmienne mogą być wiązane ze zmiennymi. Przykładowo, weźmy pod uwagę dwie zmienne niezwiązane x_1 i x_2, do których odwołują się identyfikatory `X` i `Y`. Po wykonaniu związania `X=Y` otrzymujemy sytuację z rysunku 2.14. Dwie zmienne, x_1 i x_2, są sobie równe. Na rysunku pokazano to, przedstawiając odwołanie każdej ze zmiennych do drugiej. Mówimy, że $\{x_1, x_2\}$ tworzy zbiór równoważności[4]. Zapisujemy to również jako $x_1 = x_2$. Trzy zmienne związane ze sobą zapisujemy jako $x_1 = x_2 = x_3$ lub $\{x_1, x_2, x_3\}$. W przypadku prezentacji graficznej zmienne te utworzyłyby łańcuch cykliczny. Zawsze kiedy pewna zmienna ze zbioru równoważności zostanie związana, wszystkie pozostałe zmienne widzą związanie. Na rysunku 2.15 przedstawiono rezultat wykonania operacji `X=[1 2 3]`.



RYSUNEK 2.14. Dwie zmienne związane ze sobą



RYSUNEK 2.15. Obszar po związaniu jednej ze zmiennych

[4] Z formalnego punktu widzenia te dwie zmienne tworzą klasę równoważności ze względu na relację równości.

2.2.8. Zmienne przepływu danych

W modelu deklaratywnym tworzenie zmiennych i wiązanie ich jest wykonywane oddzielnie. Można jednak zadać pytanie, co się stanie, jeżeli spróbujemy użyć zmiennej przed jej związaniem. Określamy to mianem błędu użycia zmiennej. W przypadku niektórych języków tworzenie i wiązanie zmiennej odbywa się w jednym kroku, tak więc błędy użycia nie mogę się pojawiać. Jest tak w przypadku języków programowania funkcyjnego. Inne języki pozwalają na oddzielne tworzenie i wiązanie zmiennych. Wówczas otrzymujemy następujące możliwości w przypadku wystąpienia błędu:

(1) Wykonywanie jest kontynuowane bez żadnego komunikatu o błędzie. Zawartość zmiennej jest niezdefiniowana, tzn. zawiera „śmieci" — losowe wartości znajdujące się akurat w danym obszarze pamięci. W taki sposób funkcjonuje język C++.
(2) Wykonywanie jest kontynuowane bez żadnego komunikatu o błędzie. Zmienna jest inicjalizowana wartością domyślną w momencie zadeklarowania, np. 0 w przypadku liczb całkowitych. W ten sposób funkcjonuje Java w przypadku pól w obiektach oraz strukturach danych takich jak tablice. Wartość domyślna zależy od typu.
(3) Wykonywanie zostaje zatrzymane z wyświetleniem komunikatu o błędzie (lub zgłoszeniem wyjątku). W ten sposób funkcjonuje Prolog w przypadku działań arytmetycznych.
(4) Wykonywanie nie jest możliwe, ponieważ kompilator wykrył, że istnieje ścieżka wykonania wiodąca do użycia zmiennej bez jej zainicjalizowania. W ten sposób funkcjonuje Java w przypadku zmiennych lokalnych.
(5) Wykonywanie jest wstrzymywane do momentu związania zmiennej, a później kontynuowane. W ten sposób funkcjonuje Oz w celu obsługi programowania przepływu danych.

Powyższe przypadki wymieniono według rosnącego poziomu „przyjazności". Pierwszy przypadek jest jak najmniej wskazany, gdyż różne wykonania tego samego programu mogą dawać różne wyniki. Co więcej, ze względu na fakt, że wystąpienie błędu nie jest sygnalizowane, programista nie wie nawet, kiedy sytuacja taka ma miejsce. Drugi przypadek jest nieco lepszy. Jeżeli program zawiera błąd użycia, to przynajmniej zawsze da ten sam wynik, nawet jeżeli jest on błędny. I tu programista nie jest świadomy występowania błędu.

Przypadki trzeci i czwarty mogą okazać się odpowiednie w niektórych sytuacjach. W każdym z nich program zawierający błąd użycia sygnalizuje ten fakt — w czasie wykonywania lub kompilacji. Jest to rozsądne w przypadku systemów sekwencyjnych, gdyż naprawdę ma miejsce wystąpienie błędu. Przypadek trzeci jest nieodpowiedni w przypadku systemu współbieżnego, gdyż wynik staje się niedeterministyczny — w zależności od czasów wykonania błąd może być czasem sygnalizowany, a czasem nie.

W przypadku piątym program przechodzi w stan oczekiwania do momentu, aż zmienna zostanie związana, a następnie kontynuuje działanie. Modele obliczeniowe prezentowane w niniejszej książce wykorzystują właśnie ten przypadek. Jest on nieodpowiedni dla systemu sekwencyjnego, gdyż program pozostanie w takim stanie na zawsze, jednak w przypadku systemu współbieżnego jest odpowiedni i może tam stanowić część normalnych działań związanych z operacją związania zmiennej w innym wątku. Przypadek piąty wprowadza nowy rodzaj błędu programowego, a ściśle zawieszenie, które na zawsze pozostaje w stanie oczekiwania. Przykładowo, jeżeli nazwa zmiennej została błędnie zapisana, nigdy nie zostanie ona związana. Dobry program uruchomieniowy powinien wykrywać takie sytuacje.

Zmienne deklaratywne, które powodują, że program pozostaje w stanie oczekiwania do momentu, aż zostaną one związane, określa się mianem *zmiennych przepływu danych* (ang. *dataflow variables*). Model deklaratywny wykorzystuje zmienne przepływu danych dlatego, że są one niezmiernie przydatne w zakresie programowania współbieżnego, tzn. programów wykonujących

niezależne działania. Jeżeli wykonamy dwie operacje współbieżne, na przykład `A=23` i `B=A+1`, to w przypadku piątym zawsze zostaną one zakończone poprawnie i dadzą odpowiedź `B=24`. Nie ma znaczenia, czy jako pierwsza zostanie wykonana operacja `A=23` czy `B=A+1`. W innych przypadkach nie mamy takiej gwarancji. Owa właściwość niezależności od kolejności wykonania umożliwia stosowanie współbieżności deklaratywnej, omawianej w rozdziale 4. Właśnie dlatego zmienne przepływu danych są tak przydatne.

2.3. Język modelowy

Model deklaratywny definiuje prosty język modelowy. Wszystkie programy w ramach tego modelu mogą zostać wyrażone w tym języku. Najpierw zdefiniujemy składnię i semantykę języka modelowego. Później zostanie wyjaśnione, w jaki sposób należy utworzyć pełny język na podbudowie języka modelowego.

2.3.1. Składnia

Składnia modelowa została przedstawiona w tabelach 2.1 oraz 2.2. Zaprojektowano ją z uwagą, tak aby stanowiła podzbiór składni pełnego języka, co oznacza, że wszystkie instrukcje języka modelowego są poprawnymi instrukcjami pełnego języka.

TABELA 2.1.
Deklaratywny język modelowy

⟨s⟩ ::=	
skip	Instrukcja pusta
| ⟨s⟩$_1$ ⟨s⟩$_2$	Sekwencja instrukcji
| local ⟨x⟩ in ⟨s⟩ end	Utworzenie zmiennej
| ⟨x⟩$_1$=⟨x⟩$_2$	Związanie typu zmienna-zmienna
| ⟨x⟩=⟨v⟩	Utworzenie wartości
| if ⟨x⟩ then ⟨s⟩$_1$ else ⟨s⟩$_2$ end	Instrukcja warunkowa
| case ⟨x⟩ of ⟨pattern⟩ then ⟨s⟩$_1$ else ⟨s⟩$_2$ end	Dopasowanie do wzorca
| {⟨x⟩ ⟨y⟩$_1$ ... ⟨y⟩$_n$}	Zastosowanie procedury

TABELA 2.2.
Wyrażenia wartości w deklaratywnym języku modelowym

⟨v⟩	::=	⟨number⟩ | ⟨record⟩ | ⟨procedure⟩
⟨number⟩	::=	⟨int⟩ | ⟨float⟩
⟨record⟩, ⟨pattern⟩	::=	⟨literal⟩
	|	⟨literal⟩(⟨feature⟩$_1$: ⟨x⟩$_1$ ... ⟨feature⟩$_n$: ⟨x⟩$_n$)
⟨procedure⟩	::=	proc { $ ⟨x⟩$_1$ ... ⟨x⟩$_n$} ⟨s⟩ end
⟨literal⟩	::=	⟨atom⟩ | ⟨bool⟩
⟨feature⟩	::=	⟨atom⟩ | ⟨bool⟩ | ⟨int⟩
⟨bool⟩	::=	true | false

Składnia instrukcji

W tabeli 2.1 zdefiniowano składnię dla elementu ⟨s⟩, który oznacza instrukcję. W sumie istnieje osiem instrukcji, które zostaną objaśnione poniżej.

Składnia wartości

W tabeli 2.2 zdefiniowano składnię dla elementu ⟨v⟩, który oznacza wartość. Istnieją trzy rodzaje wyrażeń wartości, oznaczających liczby, rekordy oraz procedury. W przypadku rekordów i wzorców argumenty $\langle x \rangle_1, \ldots, \langle x \rangle_n$ wszystkie muszą być odrębnymi identyfikatorami. Zapewnia to, że wszelkie wiązania typu zmienna-zmienna będą zapisywane jako jawne operacje modelowe.

Składnia identyfikatora zmiennej

W tabeli 2.1 wykorzystywane są symbole nieterminalne ⟨x⟩ oraz ⟨y⟩ w celu oznaczenia identyfikatora zmiennej. Będziemy również używać zapisu ⟨z⟩ w tym samym celu. Istnieją dwa sposoby zapisu identyfikatora zmiennej:

- Wielka litera, po której występuje zero lub więcej znaków alfanumerycznych (liter, cyfr lub znaku podkreślenia), np. `X`, `X1` lub `ToJestDlugaZmienna_NieInt`.
- Dowolny ciąg znaków drukowalnych ujętych w znaki ` (odwrotny apostrof), np. `` `to jest 25$\zmienna!` ``.

Ścisła definicja składni identyfikatora została przedstawiona w dodatku C. Wszystkie nowo deklarowane zmienne są niezwiązane, dopóki nie zostanie wykonana jakaś instrukcja. Identyfikatory wszystkich zmiennych muszą zostać zadeklarowane w sposób jawny.

2.3.2. Wartości i typy

Typ lub typ danych to zbiór wartości wraz ze zbiorem operacji wykonywanych na tych wartościach. Wartość jest „typu”, jeżeli znajduje się w zbiorze danego typu. Model deklaratywny jest typowany w tym sensie, że posiada dobrze zdefiniowany zbiór typów określanych mianem typów podstawowych. Przykładowo, programy mogą wykonywać obliczenia na liczbach całkowitych lub rekordach, które wszystkie są, odpowiednio, typu całkowitego lub rekordowego. Każda próba użycia operacji z wartościami o błędnym typie jest wykrywana przez system i powoduje zgłoszenie błędu (patrz podrozdział 2.7). Model nie nakłada żadnych innych ograniczeń na używanie typów.

Ze względu na fakt, że typy są sprawdzane, nie jest możliwe, aby zachowanie programu wykroczyło poza model, np. powodując awarię z powodu wykonania niezdefiniowanej operacji na jego wewnętrznych strukturach danych. Wciąż jednak jest możliwe, aby program spowodował zgłoszenie błędu, np. przy dzieleniu przez zero. W modelu deklaratywnym program powodujący zgłoszenie błędu jest natychmiast kończony. Model nie uwzględnia żadnych mechanizmów obsługi błędów. W podrozdziale 2.7 model deklaratywny zostanie rozszerzony o nowe pojęcie — wyjątki — służące właśnie obsłudze błędów. W modelu rozszerzonym błędy typów mogą być obsługiwane w ramach modelu.

Oprócz typów podstawowych programy mogą zawierać definicje własnych typów. Określa się je mianem *abstrakcyjnych typów danych* (ang. *abstract data types*, ADT). W rozdziale 3. i kolejnych zostanie pokazane, w jaki sposób definiuje się ADT. Oprócz ADT istnieją także inne abstrakcje danych. Przegląd różnych możliwości zawiera podrozdział 6.4.

Typy podstawowe

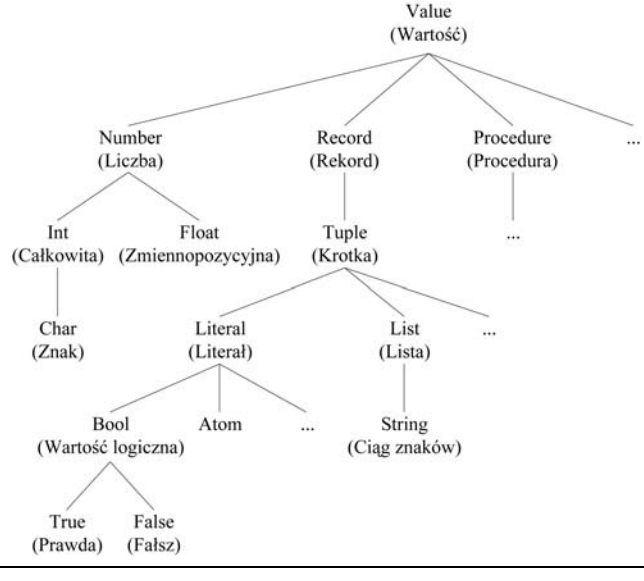
Do typów podstawowych modelu deklaratywnego należą liczby (całkowite i zmiennopozycyjne), rekordy (w tym atomy, wartości logiczne, krotki, listy i ciągi znaków) oraz procedury. W tabeli 2.2 przedstawiono ich składnię. Symbol nieterminalny ⟨v⟩ oznacza wartość częściowo skonstruowaną. W dalszej części książki zostaną zaprezentowane inne typy podstawowe, w tym porcje, funktory, komórki, słowniki, tablice, porty, klasy i obiekty. Niektóre spośród nich omówiono w dodatku B.

Dynamiczne użycie typów

Istnieją dwa główne podejścia do używania typów — dynamiczne i statyczne. W przypadku typizacji statycznej typy wszystkich zmiennych są w czasie kompilacji znane. W przypadku typizacji dynamicznej typ zmiennej jest znany dopiero po jej związaniu. Model deklaratywny jest dynamiczny pod względem użycia typów. Kompilator podejmuje próbę weryfikacji, czy wszystkie operacje wykorzystują wartości o odpowiednim typie, jednak ze względu na dynamiczny charakter użycia typów niektóre sprawdzenia z konieczności muszą zostać odłożone do czasu uruchomienia.

Hierarchia typów

Podstawowe typy modelu deklaratywnego można sklasyfikować w formie hierarchii. Przedstawiono ją na rysunku 2.16, gdzie każdy wierzchołek oznacza typ. Hierarchia została uporządkowana według zawierania się zbiorów, tzn. wszystkie wartości typu danego wierzchołka są również wartościami typu wierzchołka nadrzędnego. Przykładowo, wszystkie krotki są rekordami, a wszystkie listy są krotkami. Również wszystkie operacje danego typu są dozwolone dla jego podtypu, np. wszystkie operacje wykonywane na listach działają również w przypadku ciągów znaków. W dalszej części książki rozszerzymy tę hierarchię. Przykładowo, literały mogą być albo atomami (opisanymi poniżej), albo innym rodzajem nazw stałych (patrz punkt 3.7.5). Fragmenty, w których hierarchia jest niepełna, oznaczono jako „…”.



RYSUNEK 2.16. Hierarchia typów modelu deklaratywnego

2.3.3. Typy podstawowe

Poniżej zostaną przedstawione pewne przykłady typów podstawowych oraz sposobu ich zapisu. W dodatku B można znaleźć pełniejsze informacje na ten temat.

- *Liczby*. Liczbami są albo liczby całkowite, albo liczby zmiennopozycyjne. Przykładami liczb całkowitych są `314`, `0` oraz `~10` (minus 10). Należy zauważyć, że znak minusa jest zapisywany za pomocą tyldy (~). Przykładami liczb zmiennopozycyjnych są `1.0`, `3.4`, `2.0e2` oraz `~2.0E~2`.
- *Atomy*. Atom (wartość niepodzielna) to rodzaj stałej symbolicznej, która może być używana w obliczeniach jako pojedynczy element. Istnieje kilka różnych sposobów zapisu atomów. Atom można zapisać jako ciąg znaków rozpoczynających się od małej litery, po której występuje dowolna liczba znaków alfanumerycznych. Można go również zapisać jako dowolny ciąg znaków drukowalnych ujętych w apostrofy. Przykładami atomów są `a_osoba`, `donkeyKong3` oraz `'#### hello ####'`.
- *Wartości logiczne*. Wartością logiczną może być symbol `true` (prawda) lub `false` (fałsz).
- *Rekordy*. Rekord to złożona struktura danych. Składa się z etykiety, po której występuje zbiór par cech i identyfikatorów zmiennych. Cechami mogą być atomy, liczby całkowite lub wartości logiczne. Przykładami rekordów są: `osoba(wiek:X1 imie:X2)` (o cechach `wiek` i `imie`), `osoba(1:X1 2:X2)`, `'|'(1:H 2:T)`, `'#'(1:H 2:T)`, `nil` oraz `osoba`. Atom jest rekordem o zerowej liczbie cech.
- *Krotki*. Krotka jest rekordem, którego cechami są kolejne liczby całkowite, rozpoczynając od 1. Cechy nie muszą być w takim przypadku zapisywane. Przykładami krotek są: `osoba(1:X1 2:X2)` oraz `osoba(X1 X2)`, które oznaczają to samo.
- *Listy*. Lista jest albo atomem `nil`, albo krotką `'|'(H T)` (etykietą jest pionowa kreska), gdzie `T` jest albo niezwiązane, albo związane z listą. Taką krotkę określa się mianem pary listy lub cons. W przypadku list mamy do czynienia z lukrem składniowym:
  - Etykieta `'|'` może zostać zapisana jako operator wrostkowy, więc `H|T` oznacza to samo co `'|'(H T)`.
  - Operator `'|'` wiąże prawostronnie, więc `1|2|3|nil` oznacza to samo co `1|(2|(3|nil))`.
  - Listy kończące się atomem `nil` mogą być zapisywane przy użyciu nawiasów kwadratowych `[...]`, więc zapis `[1 2 3]` oznacza top samo co `1|2|3|nil`. Takie listy nazywa się listami pełnymi.
- *Ciągi znaków*. Ciąg znaków jest listą kodów znaków. Ciągi znaków można zapisywać przy użyciu cudzysłowów, więc zapis `"E=mc^2"` oznacza to samo co `[69 61 109 99 94 50]`.
- *Procedury*. Procedura jest wartością typu proceduralnego. Instrukcja:

  ⟨x⟩ = proc {$ ⟨y⟩1 ... ⟨y⟩n} ⟨s⟩ end

  wiąże *⟨x⟩* z nową wartością procedury. Oznacza to po prostu zadeklarowanie nowej procedury. Symbol `$` określa, że wartość procedury jest anonimowa, tzn. jest tworzona bez związania z identyfikatorem. Można wskazać lepiej znany skrót zapisu składniowego:

  proc {⟨x⟩ ⟨y⟩1 ... ⟨y⟩n} ⟨s⟩ end

  Symbol `$` został zastąpiony identyfikatorem *⟨x⟩*. Zapis taki powoduje utworzenie wartości procedury i natychmiastową próbę jej związania z *⟨x⟩*. Taki skrócony zapis jest zapewne łatwiejszy w czytaniu, ale zaciemnia rozróżnienie między utworzeniem wartości a związaniem jej z identyfikatorem.

2.3.4. Rekordy i procedury

Poniżej zostanie wyjaśnione, dlaczego wybrano rekordy i procedury jako podstawowe pojęcia języka modelowego. Niniejszy punkt jest przeznaczony dla Czytelników posiadających pewne doświadczenie programistyczne, którzy zastanawiają się, dlaczego język modelowy został zaprojektowany w taki, a nie inny sposób.

Możliwości rekordów

Rekordy to podstawowy sposób strukturyzacji danych. Stanowią one element składowy większości struktur danych, w tym list, drzew, kolejek, grafów itd., co zostanie pokazane w rozdziale 3. Rekordy odgrywają do pewnego stopnia taką rolę w większości języków programowania, jednak okazuje się, że ich potencjał sięga znacznie dalej. Owe dodatkowe możliwości ujawniają się w mniejszym lub większym stopniu w zależności od tego, jak słabo lub dobrze są obsługiwane w ramach danego języka. W celu zapewnienia maksymalnych możliwości język powinien ułatwiać ich tworzenie, rozdzielanie i manipulowanie nimi. W modelu deklaratywnym rekord jest tworzony po prostu poprzez zapisanie go przy użyciu zwięzłej składni. Jest rozdzielany za pomocą zapisanego wzorca — również przy użyciu zwięzłej składni. Wreszcie, istnieje wiele operacji manipulowania na rekordach: dodawanie, usuwanie lub wybieranie pól, konwersja na listę i odwrotnie itd. Ogólnie rzecz biorąc, języki oferujące taki poziom wsparcia dla obsługi rekordów określa się mianem języków symbolicznych.

Kiedy rekordy są dobrze obsługiwane, mogą zostać użyte do zwiększenia skuteczności wielu innych technik. W niniejszej książce szczególnie skupimy się na trzech z nich: programowaniu zorientowanym obiektowo, projektowaniu graficznych interfejsów użytkownika (GUI) oraz programowaniu komponentowym. W przypadku programowania zorientowanego obiektowo w rozdziale 7. zostanie pokazane, w jaki sposób rekordy mogą reprezentować komunikaty i nagłówki metod, za pomocą których obiekty się komunikują. W przypadku projektowania GUI w rozdziale 10. zostanie pokazane, w jaki sposób rekordy mogą reprezentować „widgety” — podstawowe bloki tworzące interfejs. W zakresie programowania komponentowego w podrozdziale 3.9 zostanie pokazane, w jaki sposób rekordy mogą reprezentować moduły pierwszorzędowe, które grupują powiązane ze sobą operacje.

Zasadność użycia procedur

Czytelnik posiadający pewne doświadczenie programistyczne może się zastanawiać, dlaczego prezentowany język modelowy zawiera jako podstawową konstrukcję procedurę. Zwolennicy programowania zorientowanego-obiektowo mogą rozważać, dlaczego nie wybrano raczej obiektów. Z kolei zwolennicy programowania funkcyjnego — dlaczego nie wybrano funkcji. Oczywiście każdy z tych wyborów wchodził w rachubę, jednak tak się nie stało. Powody są całkiem proste.

Procedury są bardziej odpowiednie niż obiekty, ponieważ są prostsze. Obiekty, jak to zostanie objaśnione w rozdziale 7., są konstrukcjami dość skomplikowanymi. Procedury są również odpowiedniejsze od funkcji, ponieważ nie zawsze definiują elementy, które zachowują się podobnie do funkcji matematycznych[5]. Przykładowo, zarówno komponenty, jak i obiekty definiujemy jako

[5] Z teoretycznego punktu widzenia procedury są „procesami”, jakich używa się w rachunkach współbieżnych, takich jak rachunek π. Argumentami są w tym przypadku kanały. W niniejszym rozdziale będą używane procesy, które są tworzone sekwencyjnie za pomocą kanałów jednokrotnych. W rozdziałach 4. i 5. zostaną zaprezentowane inne rodzaje kanałów (z sekwencjami komunikatów) oraz techniki współbieżnego składania procesów.

abstrakcje w oparciu o procedury. Ponadto procedury są elastyczne, ponieważ w ich przypadku nie są przyjmowane żadne założenia co do liczby danych wejściowych i wyjściowych. Funkcja zawsze posiada tylko jedną wartość wyjściową. Z kolei procedura może posiadać dowolną liczbę danych wejściowych i wyjściowych, w tym żadnych. Jak się okaże, kiedy będzie mowa o programowaniu wyższego rzędu w podrozdziale 3.6, procedury są oferującymi ogromne możliwości elementami budulcowymi systemów.

2.3.5. Operacje podstawowe

W tabeli 2.3 przedstawiono operacje podstawowe, które będą wykorzystywane w bieżącym i kolejnym rozdziale. Dla wielu z nich istnieje lukier składniowy, więc można je zapisywać w postaci wyrażeń bardziej zwięzłych. Przykładowo, zapis `X=A*B` to lukier składniowy dla zapisu `{Number.'*' A B X}`, gdzie `Number.'*'` jest procedurą skojarzoną z typem `Number`[6]. Wszystkie operacje można oznaczyć w pewien dłuższy sposób, np. `Value.'=='`, `Value.'<'`, `Int.'div'`, `Float.'/'`. W tabeli zawarto zapisy typu lukier składniowy tam, gdzie takie istnieją.

TABELA 2.3.
Przykłady operacji podstawowych

Operacja	Opis	Typ argumentu
`A==B`	Porównanie równości	Value
`A\=B`	Porównanie nierówności	Value
`{IsProcedure P}`	Sprawdzenie, czy procedura	Value
`A<=B`	Porównanie mniejszy niż lub równy	Number lub Atom
`A<B`	Porównanie mniejszy niż	Number lub Atom
`A>=B`	Porównanie większy niż lub równy	Number lub Atom
`A>B`	Porównanie większy niż	Number lub Atom
`A+B`	Dodawanie	Number
`A-B`	Odejmowanie	Number
`A*B`	Mnożenie	Number
`A div B`	Dzielenie	Int
`A mod B`	Modulo	Int
`A/B`	Dzielenie	Float
`{Arity R}`	Krotność	Record
`{Label R}`	Etykieta	Record
`R.F`	Wybór pola	Record

- *Arytmetyczne*. Liczby zmiennopozycyjne posiadają cztery operacje podstawowe: `+`, `-`, `*` oraz `/`. Liczby całkowite posiadają następujące operacje podstawowe: `+`, `\`, `*`, `div` oraz `mod`, gdzie `div` oznacza dzielenie całkowitoliczbowe (z obcięciem części ułamkowej), zaś `mod` oznacza operację modulo z liczby całkowitej, czyli resztę z dzielenia, na przykład: `10 mod 3` = 1.
- *Operacje na rekordach*. Trzy podstawowe operacje wykonywane na rekordach to `Arity`, `Label` oraz „.” (kropka, która oznacza wybór pola). Przykładowo, dla rekordu:

[6] Ściśle rzecz ujmując, `Number` to moduł grupujący operacje typu `Number` i zapis `Number.'*'` powoduje wybranie operacji mnożenia.

```
X=osoba(name="Grzegorz" wiek:25)
```

`{Arity X}=[age name]`, `{Label X}=osoba` oraz `X.age=25`. Wywołanie `Arity` zwraca listę, która zawiera najpierw cechy całkowitoliczbowe w porządku rosnącym, a następnie cechy atomowe w rosnącym porządku leksykograficznym.

- *Porównania*. Do logicznych funkcji porównań należą `==` oraz `\=`, które mogą służyć do sprawdzania równości dowolnych dwóch wartości. Z kolei porównania numeryczne to `=<`, `<`, `=>` oraz `>`, które służą do porównywania liczb całkowitych, zmiennopozycyjnych oraz atomów. Atomy są porównywane według porządku leksykograficznego ich reprezentacji drukowalnych. W poniższym przykładzie `Z` jest związane z maksimum wartości `X` i `Y`:

```
declare X Y Z T in
X=5 Y=10
T=(X>=Y)
if T then Z=X else Z=Y end
```

Istnieje lukier składniowy, dzięki któremu instrukcja `if` akceptuje wyrażenie jako swój warunek. Powyższy przykład można przepisać następująco:

```
declare X Y Z in
X=5 Y=10
if X>=Y then Z=X else Z=Y end
```

- *Operacje na procedurach*. Istnieją trzy operacje podstawowe na procedurach: ich definiowanie (przy użyciu instrukcji `proc`), wywoływanie (za pomocą notacji nawiasów klamrowych) oraz sprawdzanie (za pomocą funkcji `IsProcedure`) czy dana wartość jest procedurą. Wywołanie `{IsProcedure P}` zwraca wartość `true`, jeżeli `P` jest procedurą, a w przeciwnym wypadku — `false`.

W dodatku B zawarto pełniejszy zbiór operacji podstawowych.