Wstęp do programowania w języku Ada'95

Michał Morawski Antoni M. Zajączkowski

Wydanie drugie
Łódź, 2004

Poprzednia wesja niniejszego skryptu została wydana w 1999r. nakładem Wydziału FTIMS Politechniki Łódzkiej ze środków programu „Tempus"

Przedmowa

Obecnie, jednym z podstawowych elementów wykształcenia absolwentów wydziałów technicznych wyższych uczelni technicznych jest umiejętność projektowania i pisania programów komputerowych w języku wysokiego poziomu. Dotyczy to szczególnie absolwentów kierunku Informatyka i kierunków pokrewnych, takich jak Elektronika lub Elektrotechnika. Podobnie jest na Wydziale Fizyki Technicznej, Informatyki i Matematyki Stosowanej (FTIMS) Politechniki Łódzkiej. W poprzednich latach uznano, że studenci magisterskich studiów dziennych kierunku Informatyka tego wydziału powinni zdobyć umiejętność pisania programów komputerowych w kilku językach programowania wysokiego poziomu. Ustalono też, że jednym z tych języków jest język C/C++, natomiast drugim językiem, nauczanym w ramach przedmiotu *Wprowadzenie do użytkowania komputerów* (WDUK) był język Pascal, który uznano za odpowiedni język w początkowym kursie programowania jaki realizowany jest w ramach wspomnianego przedmiotu.

W latach 1998-2000 na Wydziale FTIMS dokonano reformy programu studiów dziennych wszystkich kierunków studiów, w tym kierunku Informatyka. W wyniku tych działań, finansowanych w dużym stopniu przez Unię Europejską (UE) w ramach programu Tempus, opracowano nowe programy studiów, których realizacja rozpoczęła się w semestrze zimowym roku akademickiego 2000/2001. Nowe programy dają możliwość uzyskania stopnia inżyniera, albo licencjata w przypadku kierunku Matematyka, lub stopnia magistra inżyniera, albo odpowiednio magistra. Studia inżynierskie trwają siedem semestrów, a studia magisterskie dziesięć semestrów, przy czym pierwsze cztery semestry są wspólne. Student uczący się wg nowych programów zaliczając przedmiot uzyskuje oprócz zwykłej oceny, pewną liczbę punktów kredytowych przypisaną przedmiotowi. Wprowadzony w nowych programach system punktów kredytowych ma umożliwić m.in. wymianę studentów z uczelniami krajów UE.

Prace nad nowymi programami nauczania były dobrą okazją do modernizacji treści przedmiotów realizowanych na kierunku Informatyka, a w tym przedmiotów dotyczących nauki programowania.

Niniejszy skrypt jest wynikiem części tych działań i ma służyć jako pomoc dydaktyczna studentom oraz wykładowcom realizującym przedmioty *Wprowadzenie do informatyki* (WDI), *Podstawy informatyki* (PI) i *Systemy czasu*

rzeczywistego (SCR). Pierwsze dwa przedmioty realizowane są podczas pierwszego roku studiów, odpowiednio na kierunku Informatyka oraz kierunkach Fizyka Techniczna i Matematyka, a celem ich jest m.in. nauczenie studentów podstaw programowania komputerów. Ostatni, wymieniony przedmiot przewidziano dla studentów czwartego roku studiów magisterskich kierunku Informatyka, a więc dla studentów bardziej zaawansowanych. Podczas dyskusji nad treściami programowymi zreformowanego programu nauczania uznano, że język Pascal nie spełnia w pełni współczesnych wymagań jakie powinien spełniać język programowania wysokiego poziomu stosowany do nauczania programowania w początkowej fazie studiów. Uznano też, że język C/C++ nie nadaje się do nauczania początkowego i w związku z tym powstał problem wyboru właściwego języka, przy czym założono, że najlepszym językiem będzie ten, który oparty jest o czytelną składnię Pascala i jednocześnie umożliwia pisanie programów wykonywanych w czasie rzeczywistym. Wybór nasz padł na język Ada, a dokładniej na Adę 95. Decyzję tę trzeba krótko uzasadnić. Składnia Ady jest w naszym przekonaniu bardziej czytelna od składni Pascala, a pewne, znane wady tego języka zostały w Adzie usunięte. Trzeba przy tym wspomnieć, że składnia Ady oparta jest o składnię Pascala, co ma duże znaczenie dla studentów pierwszego roku, którzy, jak zakładamy, poznali tę składnię ucząc się informatyki w szkole podstawowej i średniej. Panuje dość powszechnie przekonanie, że język Pascal jest językiem, który najlepiej nadaje się do początkowej nauki programowania wg zasad programowania strukturalnego (Wirth 1978) i celem twórcy tego języka, profesora Niklausa Wirtha z Politechniki w Zürichu było zaprojektowanie takiego języka. Od opublikowania definicji Pascala w roku 1970 obserwowaliśmy szybki rozwój informatyki, a w tym języków i inżynierii oprogramowania. Jedną z metod, która wywarła doniosły wpływ na inżynierię oprogramowania i na całą informatykę jest metoda programowania obiektowego (Coad i Nicola 1993, Martin i Odell 1997), której idee zostały zastosowane przy projektowaniu języka C++. Język Pascal w wersji wzorcowej nie zawiera konstrukcji umożliwiających stosowanie metod obiektowych, ale umożliwiają to późniejsze wersje języka Turbo Pascal opracowane przez firmę Borland oraz jej jeszcze nowszy system programowania Delphi. Z punktu widzenia nauczania programowania podstawowymi wadami Turbo Pascala, czy Delphi są, naszym zdaniem, brak mechanizmów programowania systemów czasu rzeczywistego oraz brak międzynarodowego standardu, co jest zrozumiałe, bo są to produkty komercyjne. Ada 95 jest językiem, który ma te mechanizmy i jest pierwszym językiem obiektowym dla którego opracowano i przyjęto normę międzynarodową (Barnes 1998) oraz, co jest ważnym argumentem na rzecz Ady, używany jest do nauki programowania (Smith 1996). Przy wyborze Ady kierowaliśmy się również tym, że istnieją edukacyjne wersje kompilatorów Ady 95 dostępne bezpłatnie w odpowiednich witrynach światowej sieci komputerowej, lub dołączane do niektórych podręczników Ady (Barnes 1998). Dostępność tych kompilatorów, które powinny spełniać wymagania normy języka, ma duże znaczenie przy wyborze oprogramowania przeznaczonego do celów edukacyjnych w sytuacji, gdy budżety uczelni państwowych są tak skromne jak obecnie. Cenną pomocą przy podejmowaniu decyzji okazał się artykuł Coolinga (Cooling 1996) na temat języków programowania systemów czasu rzeczywistego, w którym omówiono podstawowe wymagania jakie te języki muszą spełniać oraz na tej podstawie dokonano analizy porównawczej współcześnie stosowanych języków tego rodzaju. Z tego artykułu wynika, że w dziedzinie systemów czasu rzeczywistego

dominującymi językami są obecnie i pozostaną zapewne przez kilka następnych lat dwa języki C++ i Ada 95, przy czym w szczególnie odpowiedzialnych zastosowaniach takich jak np. systemy kontroli lotu, Ada 95 jest zdecydowanie lepsza od C++. Poza tym, jak podają (Huzar, Fryźlewicz, Dubielewicz, Hnatkowska i Waniczek 1998) niektóre firmy żądają o 20-30% mniejszej ceny za opracowanie oprogramowania, jeżeli klient zgodzi się na realizację projektu w Adzie. Interesowaliśmy się również tym, czy Ada uczona jest w innych uczelniach, w tym uczelniach partnerskich, uczestniczących w naszym programie Tempus. Miłą niespodzianką było to, że podczas naszych wizyt w Politechnice w Madrycie (Antoni M. Zajączkowski) i na Uniwersytecie w Leeds (Michał Morawski), okazało się, że również tam Ada została przyjęta jako język programowania, którego należy uczyć studentów informatyki i kursy tego języka zaczęto tam realizować. Koledzy z Politechniki w Madrycie potwierdzili też, że w wielu uniwersytetach amerykańskich uczy się Ady na pierwszym poziomie studiów (undergraduate). W Polsce Ada nie jest językiem nieznanym i wiemy, że studenci Wydziału Matematyki Uniwersytetu Łódzkiego i Politechniki Wrocławskiej przechodzą kursy tego języka.

Ostatecznie, czy nasz wybór był słuszny pokaże przyszłość i wyniki nauczania, na które niewątpliwie wpływ będzie miał sposób realizacji procesu dydaktycznego. Uważamy, że proces ten będzie łatwiejszy w realizacji, jeżeli nasi studenci oraz wykładowcy będą mieli do dyspozycji odpowiedni podręcznik. Na polskim rynku wydawniczym jest mało podręczników na temat programowania w Adzie 95. Wcześniejsza wersja tego języka, znana jako Ada 83, była opisana w dwóch książkach: Pyle'a (Pyle 1986) i Habermanna i Perry'ego (Habermann i Perry 1989), przy czym ta pierwsza nadaje się dobrze jako podręcznik do wstępnego kursu programowania, natomiast ta druga przeznaczona jest dla zaawansowanego czytelnika, znającego dobrze język Pascal. Na temat Ady 95 znamy jedynie książkę opracowaną przez zespół pracowników Politechniki Wrocławskiej kierowany przez profesora Huzara (Huzar i inni 1998). Książka ta opisuje Adę 95 w sposób kompletny, ale naszym zdaniem jest za trudna dla studentów pierwszego roku, natomiast może stanowić cenne źródło dla zaawansowanych.

Inna polska książka dotycząca Ady (Mottet i Szmuc 2002) jest ukierunkowana na konstrukcję systemów czasu rzeczywistego i również, z całą pewnością nie jest odpowiednia jako podręcznik kursu podstawowego pierwszego języka programowania.

Pisząc niniejszy skrypt postawiliśmy sobie skromniejsze i nieco inne cele niż członkowie zespołu profesora Huzara. Naszym celem było napisanie podręcznika, który choć w części, zaspokoi potrzeby związane z nauczaniem przedmiotów *Wprowadzenie do informatyki*, *Podstawy informatyki* i *Systemy czasu rzeczywistego* realizowane w nowym programie studiów dziennych na Wydziale FTIMS PŁ. Potrzeby te oraz przyjęte ograniczenie objętości skryptu, wywarły wpływ na jego treść. Skrypt nasz można podzielić na dwie części: pierwsza część, napisana przez Antoniego Zajączkowskiego, obejmuje rozdziały 1 — 9 i zawiera materiał przewidziany do omówienia w ramach przedmiotu *Wprowadzenie do informatyki*, natomiast część druga, opracowana przez Michała Morawskiego, obejmuje rozdziały 10 — 14 i przeznaczona jest jako pomoc dla studentów bardziej zaawansowanych, a przede wszystkim dla tych, którzy przechodzą kurs *Systemy czasu rzeczywistego*. Warto tu zaznaczyć, że Ada 95

jest językiem uniwersalnym, stosowanym w wielu dziedzinach programowania i potrzeba spełnienia wymagań tych, różnych dziedzin sprawiła, że jest to język obszerny. Nasz skrypt nie opisuje całego języka, a jedynie te jego konstrukcje i własności, które uznaliśmy za najważniejsze przy realizacji wymienionych przedmiotów. Zdajemy sobie sprawę, że pewne zagadnienia nie zostały omówione, a inne omawiane są pobieżnie. Świadomi tego staraliśmy się podawać źródła, gdzie można znaleźć pełniejsze omówienie tych zagadnień, przy czym spis literatury nie jest kompletny i podaliśmy wyłącznie źródła, z których korzystaliśmy pisząc skrypt oraz wymieniliśmy niektóre podręczniki programowania wydane w Polsce, których lektura może uzupełnić wiedzę zainteresowanego Czytelnika.

Skrypt nasz zawiera zapewne błędy i niedociągnięcia, które jeżeli zostaną zauważone przez uważną Czytelniczkę lub Czytelnika, niech nie pozostaną wyłącznie do ich wiadomości, ale zostaną przekazane autorom, którzy przyjmą merytoryczną krytykę z pokorą i postarają się w przyszłości uwzględnić uwagi krytyczne w następnych wersjach skryptu. Ewentualne uwagi prosimy kierować na adres:

Michał Morawski, dr inż.,
Antoni M. Zajączkowski, dr inż.,
Samodzielny Zakład Sieci Komputerowych
Wydział FTIMS, Politechnika Łódzka,
ul. Stefanowskiego 18/22, 90-537 Łódź,
lub na jeden z adresów poczty elektronicznej:
morawski@zsk.p.lodz.pl
zajaczko@zsk.p.lodz.pl

Michał Morawski i Antoni M. Zajączkowski

Spis treści

Część I

Podstawy Ady

Rozdział 1

Wprowadzenie

Informatyka jest zespołem dyscyplin naukowych i technicznych zajmujących się automatycznym przetwarzaniem, przechowywaniem i przesyłaniem informacji oraz projektowaniem, budową i eksploatacją niezbędnych do tego celu środków technicznych (Małecki, Arendt, Bryszewski i Krasiukianis 1997). Współcześnie, do automatycznego przetwarzania informacji najczęściej stosuje się różnego rodzaju komputery, które przetwarzają informacje w postaci cyfrowej. Ponieważ przyjmuje się tu, że komputery nie oceniają wartości przetwarzanych informacji, czasami lepiej mówić o komputerowym przetwarzaniu danych. Dane, które wprowadzane są do komputera w celu ich przetworzenia nazywane są *danymi wejściowymi*, natomiast dane otrzymywane w efekcie przetwarzania nazywane są *danymi wyjściowymi*. Podczas procesu przetwarzania komputer tworzy i operuje na *danych pośrednich*.

Cechą charakterystyczną komputerów jest to, że składają się one ze sprzętu i oprogramowania. Możliwość tworzenia różnego rodzaju programów przez różnych ludzi i wykonywania tych programów na tym samym sprzęcie zadecydowała m. in. o szybkim rozwoju informatyki i jej zastosowań. Dalej projektowanie i tworzenie programów komputerowych nazywamy programowaniem komputerów, albo krótko *programowaniem*.

1.1 Pojęcia podstawowe programowania[1]

Akcja jest zdarzeniem o skończonym czasie trwania i o zamierzonym, dobrze określonym skutku, czyli *efekcie*. Każda akcja wymaga istnienia obiektu, na którym jest wykonywana, przy czym wynik akcji jest rozpoznawany przez zmiany *stanu* tego obiektu. Akcja musi być opisana w terminach pewnego języka, albo układu formuł. Opis akcji nazywa się *instrukcją*. Akcja złożona z części nazywana jest *procesem obliczeniowym, albo obliczeniem*. Dalej stosuje się też nazwę *proces*. Proces, którego części następują po sobie kolejno w czasie nazywamy *procesem sekwencyjnym*.

[1] W oparciu o (Wirth 1978, Dale, Weems i McCormick 2000)

1

Ciąg instrukcji opisujący proces obliczeniowy nazywany jest *programem*. Program składa się więc ze zbioru instrukcji, przy czym tekstowej kolejności instrukcji nie musi w ogólnym przypadku odpowiadać kolejność w czasie odpowiadających im akcji. *Procesor* jest urządzeniem, które wykonuje pewne akcje zgodnie z instrukcjami i obliczenia zgodnie z programami. Warto w tym miejscu dodać, że każda akcja wymaga pewnej, zależnej od procesora ilości *pracy*. Ilość ta może być mierzona czasem trwania akcji, co może być uznane za miarę kosztu działania procesora.

W ogólnym przypadku programy mają sens bez odwoływania się do konkretnych procesorów pod warunkiem, że sposób zapisu programu jest dokładnie określony. Oznacza to, że programista nie musi interesować się procesorem dopóty, dopóki jest pewien, że procesor rozumie język, w którym wyrażono program. Należy więc znać rodzaj instrukcji, które dany procesor może wykonać i do niego dopasować język, w którym wyraża się instrukcje.

Następnym podstawowym pojęciem w dziedzinie programowania jest pojęcie algorytmu. *Algorytm* jest to specyfikacja ciągu instrukcji, które w wyniku działania na wejściowy zbiór danych dają w skończonym czasie wyjściowy zbiór danych. Inaczej (Małecki i inni 1997), algorytmem nazywa się sposób rozwiązania pewnego problemu podany w formie ciągu instrukcji określającego skończoną liczbę operacji oraz kolejność, w jakiej te operacje powinny być wykonywane. Istotną cechą algorytmów stosowanych w programowaniu komputerów jest to, że kończą się w ograniczonym przedziale czasu.

Dalej będą interesować nas programy przeznaczone do wykonania przez komputery. Programy takie wyrażane są w *językach programowania komputerów*, które nazywane są krótko *językami programowania*.

Algorytm realizowany przez komputer, czyli zapisany w postaci programu, musi być przedstawiony w formie zrozumiałej dla komputera jak i dla programisty. Język programowania służy do wyrażenia programu w formie zrozumiałej dla obydwu stron. W ten sposób dochodzimy do następującego określenia: *programem* nazywamy algorytm zapisany w języku programowania.

Współczesne komputery są zazwyczaj urządzeniami cyfrowymi i takie komputery nazywamy niekiedy *komputerami cyfrowymi*. Oznacza to, że dane przetwarzane przez te komputery reprezentowane są jako pewne ciągi skończone, których wyrazy mogą przyjmować tylko dwie wartości, oznaczane czasami przez 0 i 1. Dane takie nazywamy *danymi binarnymi* i mówimy, że komputer (cyfrowy) przetwarza dane binarne, lub dane wyrażone w kodzie binarnym. Pierwsze programy komputerowe również wyrażano w kodzie binarnym. Powodowało to, że programy te były zrozumiałe jedynie dla specjalistów znających kody binarne poszczególnych rozkazów procesora, który miał wykonać program oraz niezmiernie utrudniało usuwanie błędów i modyfikowanie takich programów. Dodatkową wadą zapisywania programów w kodzie binarnym, był całkowity brak przenośności programów na inne procesory. Wynikało to z tego, że każdy procesor ma ściśle określoną listę rozkazów, którym przypisane są ich unikalne kody binarne. W początkowym okresie rozwoju informatyki komputery cyfrowe nazywano też maszynami cyfrowymi i stąd programy zapisane w kodzie binarnym nazywamy programami wyrażonymi w *języku maszynowym*, albo *kodzie maszynowym*. Dokładniej, *językiem maszynowym* (Dale i inni 2000) nazywamy język składający się z rozkazów procesora wyrażonych

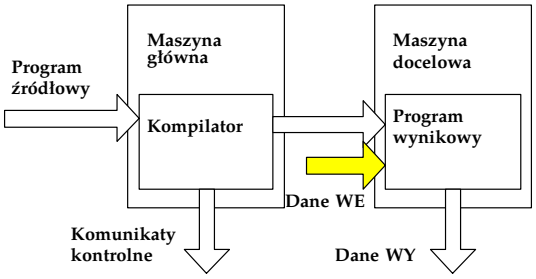
w kodzie binarnym. Wymienione wady programowania komputerów w języku maszynowym spowodowały, że rozkazy procesorów zaczęto oznaczać tzw. *mnemonikami* (np. rozkaz dodawania oznaczono przez ADD, a odejmowania przez SUB). W ten sposób stworzono języki programowania, nazywane językami asemblera. Inaczej mówiąc (Dale, Weems, McCormick, 2000) *językiem asemblera* nazywamy język programowania, w którym rozkazom języka maszynowego danego procesora jednoznacznie przyporządkowuje się mnemoniki. Jest oczywiste, że programy pisane w języku asemblera są znacznie bardziej czytelne od pisanych w języku maszynowym, ale mimo tego posiadają istotne wady: są zrozumiałe jedynie dla programistów znających język asemblera konkretnego procesora i przeznaczone są jedynie na procesory, dla których dany język asemblera opracowano. Stąd wynika, że przenośność oprogramowania zapisanego w języku asemblera ograniczona jest do pewnego konkretnego procesora, albo konkretnej rodziny procesorów (np. Intel 80*86). Ponieważ procesor może wykonać program wyrażony jedynie w kodzie maszynowym, program zapisany w języku asemblera musi być przetłumaczony na odpowiadający mu program wyrażony w języku maszynowym. Do tego celu służą odpowiednie programy, nazywane *asemblerami*. Osiągnięto w ten sposób automatyzację procesu generowania kodu maszynowego, co znacznie przyspieszyło rozwój informatyki. Dalszym krokiem było opracowanie języków programowania nazywanych *językami wysokiego poziomu*, w odróżnieniu od języków asemblera, nazywanych *językami niskiego poziomu*. Z założenia, języki wysokiego poziomu umożliwiają pisanie programów niezależnych od konkretnej maszyny przez którą mają być wykonywane. Pierwszym takim językiem jest opracowany przez firmę IBM język FORTRAN, którego ulepszone wersje są do dzisiaj stosowane w niektórych środowiskach programistów, a szczególnie chętnie FORTRAN stosuje się w środowisku związanym z naukami fizycznymi. Znanymi językami wysokiego poziomu są też COBOL, Algol, Pascal, Modula 2, Ada 83, C++, Java i Ada 95. Opracowanie jezyków wysokiego poziomu, które umożliwiają zapis programu w formie odzwierciedlającej sposób rozwiązania problemu informatycznego przez człowieka, a nie maszynę cyfrową, spowodowało, że programy komputerowe mogą pisać ludzie nie znający dokładnie komputerów, które mają te programy wykonywać. Dzięki temu rozwój informatyki uległ dalszemu przyspieszeniu, a skutki tego rozwoju (pozytywne i negatywne) wszyscy obserwują i odczuwają i będą zapewne obserwować i odczuwać w przyszłości. Dalej, programy zapisane w języku programowania wysokiego poziomu, będziemy nazywali krótko programami.

1.2 Kompilacja i interpretacja[2]

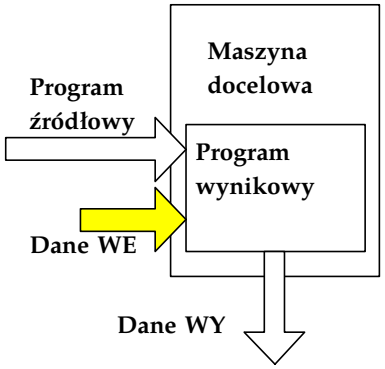
Program napisany w języku programowania nazywamy *programem źródłowym*. Program ten należy przetłumaczyć na równoważny program wyrażony w *języku maszynowym* tego komputera, na którym ma być wykonany. Po takim tłumaczeniu dostajemy *program wynikowy*, a komputer na którym może on zostać wykonany nazywamy *maszyną docelową*.

Tłumaczenia programu źródłowego na program wynikowy dokonuje program nazywany *kompilatorem*, albo *translatorem*. Kompilator działa na maszynie

[2] W oparciu o (Marcotty i Ledgard 1991)



Rysunek 1.1: Kompilacja przechodnia



Rysunek 1.2: Interpretacja

głównej, która zazwyczaj, ale nie zawsze jest maszyną docelową. Jeżeli kompilacja jest wykonywana na maszynie głównej, a program wynikowy jest przeznaczony dla innej maszyny, to używa się nazwy *kompilator przechodni*, albo *skrośny* (ang. cross compiler). Mechanizm ten pokazano schematycznie na rysunku 1.1.

Często zamiast kompilacji stosuje się *interpretację*. W takim przypadku program źródłowy nie jest tłumaczony, ale jest wykonywany instrukcja po instrukcji przez *interpretator* działający na maszynie docelowej. Interpretacja polega na sprawdzeniu instrukcji i wykonaniu akcji określonych przez tę instrukcję. Typowym przypadkiem jest wywołanie odpowiednich podprogramów. Schemat interpretacji pokazano na rysunku 1.2.

Łatwo przewidzieć, że interpretacja jest procesem znacznie wolniejszym. Między kompilacją i interpretacją nie ma ostrej granicy i często stosuje się połączenie tych dwóch technik.

Dalej będzie nas interesowało pisanie programów źródłowych w nowoczesnym języku programowania Ada 95, który będziemy nazywać krótko Adą.

1.3 Ogólna struktura programu w Adzie

Na wstępie podamy jeszcze jedną definicję języka programowania, która określa język programowania całkowicie w oderwaniu od sprzętu komputerowego

wykonującego programy. *Językiem programowania* nazywamy układ reguł, symboli i słów kluczowych (specjalnych) używanych do zapisywania programu (Dale i inni 2000). Wspomniane reguły dotyczą składni i semantyki języka, przy czym *składnia* jest to zbiór reguł określających jak zapisywane są poprawne instrukcje – konstrukcje w języku programowania, natomiast *semantyka* jest to zbiór reguł określających jakie znaczenie mają instrukcje/konstrukcje zapisane w języku programowania. Składnia określa więc, jakie kombinacje liter, liczb i symboli mogą być użyte w języku programowania. Te kombinacje są następnie używane do zapisu programu. Jeżeli kompilator ma dokonać automatycznego tłumaczenia programu na kod maszynowy, składnia programu musi być określona jednoznacznie. W tym celu, do opisu składni stosuje się pewien język formalny, nazywany *metajęzykiem*. Inaczej mówiąc, metajęzykiem nazywamy język służący do definicji innego języka (Dale i inni 2000). Do definiowania języków programowania wysokiego poziomu opracowano specjalną notację, nazywaną *notacją Backusa-Naura* (BNF - Backus-Naur Form) od nazwisk jej twórców.

Składnia Ady zapisana jest formalnie przy użyciu *rozszerzonej notacji Backusa-Naura* (EBNF - Extended Backus-Naur Form) i składnię tę zawiera *Ada Reference Manual* (Intermetrics Inc. 1995*b*) firmowany przez Intermetrics Inc. oraz książki (Barnes 1998, Dale i inni 2000). Jeżeli stosowana jest wspomniana notacja, to do określania elementów składni języka stosuje się następujące symbole specjalne:

::=	jest określone *jako*,
[ ]	element w nawiasie prostokątnym jest opcjonalny,
{ }	element w nawiasie sześciennym może być powtórzony 0 lub więcej razy,
|	lub, symbol służy do oddzielania elementów zamiennych.

W celu ilustracji jak stosuje się te symbole do określania elementów języka, podajemy uproszczone definicje cyfry (digit) i liczby całkowitej (integer):

```
digit ::= 0|1|2|3|4|5|6|7|8|9
integer ::= digit{[_]digit}
```

Według takiej metody zdefiniowana jest cała składnia Ady, przy czym należy podkreślić, że nieprzygotowana osoba może mieć znaczne trudności ze zrozumieniem konkretnego elementu składni, ponieważ w przypadku bardziej złożonych definicji, a takich jest w Adzie dużo, definicja taka składa się z elementów zdefiniowanych przy użyciu innych, bardziej prostych elementów i zanim napotkamy na elementy, których definicje już znamy, musimy poznać i zrozumieć inne definicje, które w naszym, konkretnym przypadku nie interesują nas. W związku z tym, lepiej i łatwiej uczyć się języka poznając najpierw najprostsze konstrukcje i w miarę wzbogacania swej wiedzy i nabywania doświadczenia m.in. przez rozwiązywanie zadań i pisanie programów, dochodzić do zagadnień bardziej skomplikowanych. Takie podejście do nauki programowania przyjęto w naszym podręczniku, przy czym rozszerzona notacja Backusa-Naura jest stosowana, często w uproszczeniu, do definiowania niektórych konstrukcji Ady. Używamy tej notacji tam, gdzie łatwo zrozumieć definiowaną konstrukcję i gdy

1

notacja ta umożliwia najlepsze, naszym zdaniem, przedstawienie definiowanej struktury. Poza tym, student nauk ścisłych, takich jak matematyka, fizyka i informatyka powinien możliwie wcześnie poznawać narzędzia i metody ścisłego definiowania pojęć stosowanych w studiowanych dyscyplinach.

Pierwszym zastosowaniem wspomnianej notacji jest przedstawienie struktury programu w Adzie. Program taki określony jest następujaco:

```
ada_program ::= [context_clause]
  procedure program_name is
    [declarative_part]
  begin
    sequence_of_statements
  end program_name;
```

Poszczególne konstrukcje występujące w podanej definicji nazywamy odpowiednio:

context_clause	listą importową,
program_name	nazwą, albo identyfikatorem programu,
declarative_part	częścią deklaracyjną programu,
sequence_of_statements	ciągiem instrukcji, albo częścią operacyjną programu,

natomiast słowa procedure, is, begin i end należą do zbioru słów kluczowych Ady, których znaczenie poznamy czytając następne rozdziały. Należy zwrócić uwagę na to, że lista importowa oraz część deklaracyjna są opcjonalnymi elementami programu, natomiast część operacyjna musi w programie wystąpić.

W celu zilustrowania jak wygląda konkretny program w Adzie napiszemy program obliczający pierwiastki równania kwadratowego

$$Ax^2 + Bx + C = 0,\ A, B, C \in \Re. \tag{1.1}$$

Wiadomo z algebry, że jeżeli $A \neq 0$, to rozwiązanie problemu zaczynamy od obliczenia wyróżnika

$$W = B^2 - 4AC \tag{1.2}$$

W zależności od znaku wyróżnika mamy dwa przypadki

1. Jeżeli $W < 0$, to pierwiastkami równania (1.1) jest para sprzężonych liczb zespolonych

$$x_1 = \frac{-B - j\sqrt{|W|}}{2A} \tag{1.3}$$

$$x_2 = \frac{-B + j\sqrt{|W|}}{2A} \tag{1.4}$$

przy czym j oznacza jedynkę urojoną (Trajdos, 1998).

2. Jeżeli $W \geqslant 0$, to mamy dwa pierwiastki rzeczywiste dane wzorami

$$x_1 = \frac{-B - \sqrt{W}}{2A} \tag{1.5}$$

$$x_2 = \frac{-B + \sqrt{W}}{2A} \tag{1.6}$$

Nietrudno widzieć, że jeżeli $W = 0$, to $x_1 = x_2$ i mówimy o pierwiastku podwójnym.

Przy pisaniu programu założymy, że użytkownik programu powinien wprowadzać wartości współczynników A, B, C przy pomocy klawiatury i po obliczeniu pierwiastków wyniki powinny zostać wypisane na ekranie.

Poza tym, ponieważ jest to nasz pierwszy program, nie będziemy określali własnego typu danych służącego do reprezentacji liczb zespolonych i wykorzystamy standardowy typ Float służący do reprezentacji liczb rzeczywistych.

Krótka analiza wzorów na pierwiastki równania (1.1) pozwala zauważyć, że dogodnie jest wprowadzić dwie zmienne pomocnicze Re, Im, których wartości obliczamy wg wzorów

$$Re = \frac{B}{2A}; \qquad Im = \frac{\sqrt{|W|}}{2A}. \tag{1.7}$$

Pozwala to na zapisanie wzorów na pierwiastki w postaci:

1. Pierwiastki zespolone: $x_1 = -Re - j * Im$, $x_2 = -Re + j * Im$,
2. Pierwiastki rzeczywiste: $x_1 = -Re - Im$, $x_2 = -Re + Im$.

Odpowiedni tekst programu w Adzie może mieć następującą formę:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Float_IO; use Ada.Float_IO;
3  with Ada.Numerics.Elementary_Functions;
4  use Ada.Numerics.Elementary_Functions;
5
6  procedure Pierwiastki_Trojmianu is
7     -- Program oblicza pierwiastki równania ax**2+bx+c = 0
8     -- o współczynnikach rzeczywistych
9     A, B, C : Float; -- współczynniki trójmianu
10    W : Float; -- wyróżnik
11    Re, Im : Float; -- wyniki pośrednie
12 begin
13    Put ("Podaj wspolczynnik przy x**2 : "); Get (A);
14    Put ("Podaj wspolczynnik przy x : "); Get (B);
15    Put ("Podaj wyraz wolny : "); Get (C); New_Line;
16    if A = 0.0 then
17       Put_Line ("To nie jest wielomian drugiego stopnia");
18    else
```

```
19          W := B*B - 4.0*A*C;
20          Re := B/(2.0*A); Im := Sqrt(Abs(W))/(2.0*A);
21          Put ("Wyroznik = "); Put (W); New_Line;
22          if W < 0.0 then
23              Put_Line ("Pierwiaski zespolone:");
24              Put ("x1 = "); Put (-Re); Put(" -j "); Put (Im); Put (" ");
25              Put ("x2 = "); Put (-Re); Put(" +j "); Put (Im); New_Line;
26          else
27              Put_Line ("Pierwiastki rzeczywiste:");
28              Put ("x1 = "); Put (-Re-Im); Put (" ");
29              Put ("x2 = "); Put (-Re+Im);
30          end if;
31      end if;
32  end Pierwiastki_Trojmianu;
```

W tekście programu występują słowa napisane pogrubioną czcionką. Słowa te należą do zbioru *słów zastrzeżonych*, albo *kluczowych* Ady, który podano w następnym podrozdziale. Na początku programu znajdują się linie zawierające informacje z których pakietów (modułów) bibliotecznych importowane są funkcje i procedury używane w programie. Są to trzy pakiety:

```
Ada.Text_IO;
Ada.Float_IO;
Ada.Numerics.Elementary_Functions;
```

W następnej linii mamy nagłówek programu procedure Pierwiastki_Trojmianu w którym programowi przypisano identyfikator Pierwiastki_Trojmianu, czyli jego nazwę. Po nagłówku mamy dwie linie, z których każda poprzedzona jest dwoma myślnikami --. Linie te zawierają *komentarze*, pisane tutaj kursywą, które dostarczają czytelnikowi programu odpowiednich wyjaśnień. Podczas kompilacji komentarze są ignorowane, a więc nie mają żadnego wpływu na działanie programu. Należy jednak podkreślić, że dobre komentarze są nieocenionym narzędziem dokumentowania oprogramowania (Van Tassel, 1982). Po komentarzach występują linie z deklaracjami zmiennych. Deklaracje te służą nazwaniu zmiennych i określeniu ich typów. W naszym programie mamy trzy zmienne A, B, C, które oznaczają współczynniki rozwiązywanego równania i ich wartości są danymi wejściowymi programu. Zmienne W, Re, Im służą do przechowywania danych pośrednich. Danym wyjściowym, czyli wartościom pierwiastków równania nie odpowiadają zmienne, chociaż można tak zmodyfikować program, że takie zmienne się pojawią. Deklaracja takich zmiennych byłaby potrzebna, gdyby procedura obliczania pierwiastków była częścią jakiegoś większego programu. Ponieważ w naszym prostym przykładzie zmienne mają bardzo krótkie nazwy - *identyfikatory zmiennych*, niewiele mówiące o roli tych zmiennych w programie, po deklaracjach dopisano komentarze dostarczające potrzebnych informacji. Wszystkie zmienne są typu zmiennoprzecinkowego, służącego do reprezentacji liczb rzeczywistych i typ ten w Adzie nosi nazwę Float. Po deklaracjach zmiennych zaczyna się część operacyjna programu, co zaznacza się słowem zastrzeżonym begin. Część tę kończy słowo zastrzeżone end, po którym musi wystąpić identyfikator programu zakończony średnikiem. Pomiędzy tymi dwoma słowami znajdują się instrukcje opisujące działanie programu. Najpierw na ekranie pojawia się *napis*:

```
Podaj wspolczynnik przy x**2 :
```

na który należy odpowiedzieć wprowadzając przy pomocy klawiatury wartość zmiennej A. Napis wypisywany jest przez wywołanie *procedury* Put, natomiast wywołanie procedury Get powoduje wczytanie wartości zmiennej A. Zauważmy, że podobnie jak w matematyce, w nawiasach okrągłych podane są odpowiednie *parametry aktualne* tych procedur. Analogicznie wprowadza się wartości innych współczynników. Ze wzorów (1.3) wynika, że jeżeli wprowadzimy wartość $A = 0.0$, to istnieje niebezpieczeństwo dzielenia przez zero. Program zabezpieczony jest przed taką sytuacją przez zastosowanie *instrukcji warunkowej* if, która rozdziela działanie programu na dwie ścieżki. Jeżeli prawdziwa jest relacja $A = 0.0$, to wykonywane są instrukcje pomiędzy słowami then i else tzn. wypisywany jest komunikat:

```
To nie jest wielomian drugiego stopnia
```

i program kończy się. W przeciwnym przypadku wykonywany jest ciąg instrukcji pomiędzy słowami else, i end if. Najpierw obliczana jest wartość wyróżnika równania, czyli wykonywana jest *instrukcja przypisania* zmiennej W wartości *wyrażenia* $B * B - 4.0 * A * C$. Instrukcja przypisania oznaczona jest symbolem $:=$, natomiast $*$ i $-$ oznaczają odpowiednio *operatory* mnożenia i odejmowania. Kolejno obliczane są wartości zmiennych pomocniczych Re oraz Im, przy czym znak / oznacza operator dzielenia, a Sqrt i Abs oznaczają funkcje elementarne, importowane z pakietu Ada.Numerics.Elementary_Functions obliczające odpowiednio pierwiastek kwadratowy i wartość bezwzględną. Po nadaniu wartości zmiennym Re i Im, wypisywana jest wartość wyróżnika i dalej mamy wewnętrzną, w stosunku do poprzednio omówionej, instrukcję if (opisaną dokładniej w rozdziale 4.5), która służy do rozróżnienia przypadków ujemnego i nieujemnego wyróżnika. W zależności od wartości zmiennej W, wypisywane są wartości pierwiastków zespolonych, albo rzeczywistych i program kończy się.

Rozdział 2

Zapisywanie treści programów

W Adzie programy zapisywane są jako linie tekstu utworzonego z następujących znaków:

1. małych i dużych liter alfabetu łącznie z literami akcentowanymi
2. cyfr dziesiętnych 0..9
3. znaków specjalnych # & ( ) * + , − . / : ; < = > _ | " '
4. znaku odstępu czyli spacji

2.1 Identyfikatory i słowa zastrzeżone

Identyfikatory służą do jednoznacznego wyróżniania obiektów w programie tzn. do ich nazywania. W Adzie *identyfikatorami* są skończone, ale dowolnie długie ciągi liter, cyfr i znaku podkreślenia (podkreślnika) zaczynające się od litery, przy czym należy pamiętać, że dwa podkreślniki obok siebie są niedopuszczalne oraz identyfikatorami nie mogą być następujące *słowa zastrzeżone*:

abort	abs	abstract	accept	access
aliased	all	and	array	at
begin	body	case	constant	declare
delay	delta	digits	do	else
elsif	end	entry	exception	exit
for	function	generic	goto	if
in	is	limited	loop	mod
new	not	null	of	or
others	out	package	pragma	private
procedure	protected	raise	range	record
rem	renames	requeue	return	reverse
select	separate	subtype	tagged	task
terminate	then	type	until	use
when	while	with	xor	

Formalnie, czyli w rozszerzonej notacji Backusa-Naura identyfikatory określone są następująco:

identifier ::= letter{[_]letter_or_digit}
letter_or_digit ::= letter | digit
letter ::= upper_case_letter | lower_case_letter | accented_letter,

przy czym poniżej podajemy polskie nazwy tych konstrukcji: identifier – identyfikator, letter – litera, digit – cyfra (zdefiniowana w poprzednim rozdziale), upper_case_letter – wielka litera alfabetu łacińskiego, lower_case_letter – mała litera alfabetu łacińskiego, accented_letter – litera akcentowana.

Nietrudno zauważyć, że analizując tę definicję możemy podać zasady tworzenia poprawnych identyfikatorów np. jedną z nich jest to, że identyfikator nie musi zawierać podkreślnika. Ogólnie można powiedzieć, że identyfikatory występują w dwóch miejscach programu. Pierwszym miejscem jest obszar definiowania – deklarowania obiektu. W obszarze tym obiektowi przypisywana jest nazwa tego obiektu, czyli jego identyfikator. Drugim miejscem jest obszar wykorzystywania obiektu. W obszarze tym obiekt jest identyfikowany przez swoją wcześniej określoną nazwę – identyfikator. Dochodzimy w ten sposób do opisowego określenia identyfikatora (Dale i inni 2000): *Identyfikator* jest nazwą procesu, albo obiektu i służy do odwoływania się do tego procesu, albo obiektu.

Należy pamiętać, że w Adzie nie są rozróżniane wielkie i małe litery. Dla przykładu ciągi znaków

Ada_95, ADA_95, ada_95

oznaczają ten sam identyfikator.

Poprawne są też identyfikatory

Jürgen, Æleonora[1]

natomiast wyjaśnienie dlaczego następujące ciągi znaków nie są identyfikatorami w Adzie pozostawiamy czytelnikowi

Predkosc—katowa,
Prędkosc__Katowa,
Predkosc Katowa,
Jas&Malgosia,
44dziad,
begin.

Jeżeli konstruujemy formalnie poprawne identyfikatory, to nie oznacza to jeszcze, że stosujemy dobry styl programowania. W przypadku identyfikatorów warto przestrzegać następujących zaleceń stylistycznych:

1. Używaj identyfikatorów zrozumiałych dla czytającego Twój program,
2. Staraj się używać identyfikatorów określających jaką rolę pełnią w programie obiekty, albo procesy, którym przypisujesz identyfikatory,
3. Używaj tej samej postaci identyfikatora w całym programie.

[1] Jednakże nie zalecamy używania takich identyfikatorów

Ostatnie zalecenie wynika z tego, że w Adzie np. identyfikatory MOJ_PROGRAM, mOJ_pROGRAM i Moj_Program są tymi samymi identyfikatorami[2], natomiast czytanie i rozumienie programu, w którym występuje dowolna z podanych form, w zależności od nastroju, lub fantazji programisty, może być utrudnione, a nawet denerwujące.

2.2 Liczby

Ciągi znaków służące w Adzie do zapisywania liczb mają dwie postacie. Pierwsza służy do przedstawiania liczb całkowitych, a druga liczb rzeczywistych. Cechą która różni te dwa zapisy jest to, że ciąg znaków reprezentujący liczbę rzeczywistą zawiera kropkę dziesiętną podczas gdy ciąg reprezentujący liczbę całkowitą nigdy nie zawiera tej kropki. Warto tu podkreślić, że liczby całkowite są reprezentowane dokładnie, natomiast komputerowa reprezentacja liczby rzeczywistej jest w ogólnym przypadku tylko pewną aproksymacją.

Dobrym przykładem jest liczba π, którą może reprezentować ciąg 3.1415926536. Jest oczywiste, że łatwo znaleźć lepsze przybliżenie, ale nigdy nie będziemy mieli dokładnej reprezentacji tej liczby.

W Adzie liczby można zapisywać przy dowolnej podstawie od 2 do 16, przy czym symbolami reprezentującymi cyfry 10, 11, 12, 13, 14, 15 są odpowiednio wielkie litery A, B, C, D, E, F.

Niech będzie dana liczba 8097. Przy podstawie 16 liczbę tę reprezentuje ciąg 16#1FA1#, natomiast przy postawach 8 i 2 liczbę tę reprezentują ciągi: 8#17641# i 2#1111110100001#. Oczywiście można jawnie pisać podstawę 10, tzn. poprawny jest zapis 10#8097#. Sytuację tę ilustruje prosty program:

```
1  with Ada.Text_Io;
2  with Ada.Integer_Text_Io;
3  procedure Podstawy_Integer is
4      Liczba_10, Liczba_2, Liczba_16 : Integer;
5  begin
6      Liczba_10 := 10#8097#;
7      Liczba_2 := 2#1111110100001#;
8      Liczba_16 := 16#1FA1#;
9      if (Liczba_10 = Liczba_2) and (Liczba_2 = Liczba_16) then
10         Ada.Text_Io.Put ("To jest ta sama liczba = ");
11         Ada.Integer_Text_Io.Put (Liczba_2);
12     end if;
13 end Podstawy_Integer;
```

Inaczej niż w wielu innych językach, w Adzie liczby całkowite można zapisywać w postaci wykładniczej, przy czym wykładnik musi być liczbą całkowitą nieujemną. Liczba 2000 może być zapisana w następujących, równoważnych postaciach:

2000E+0, 2e3, 20E+2, 200e+1,

[2] chociaż używając kompilatora GNAT dostępnego na licencji GNU, używając opcji kompilatora żądających sprawdzania stylu, tj. «−gnatf −gnaty −gnato −gnata −gnatn −gnatwu» kompilator uzna, że identyfikatory napisane w postaci niż w ich ta, która służyła ich deklaracji, są błędne.

natomiast nie wolno pisać:

2000E−0, 20000e−1.

2

Postać wykładnicza może być stosowana do zapisywania liczb całkowitych przy dowolnej, dopuszczalnej w Adzie podstawie. Należy przy tym pamiętać, że wykładnik podaje wartość potęgi podstawy, przy której wyrażono liczbę. Podobnie jak podstawa, również wykładnik jest wyrażony w systemie dziesiętnym. Mamy w związku z tym:

$16\#B\#e+2 = 11 * 16^2 = 11 * 256 = 2816,$

$2\#111\#E5 = 7 * 2^5 = 7 * 32 = 224.$

Liczby rzeczywiste najczęściej przedstawiamy w postaci ciągu cyfr dziesiętnych i kropki dziesiętnej, która jest wewnętrznym znakiem ciągu. Oznacza to, że przed i po kropce musi być co najmniej jedna cyfra. Częściej niż w przypadku liczb całkowitych stosowana jest postać wykładnicza liczb rzeczywistych. Równoważne sposoby zapisu liczby rzeczywistej 1.732050807569, która jest przybliżeniem $\sqrt{3}$, mogą być następujące:

0.1732050807569e+1, 173.2050807569E−2,

1.732050807569e0, 1.732_050_807_569.

Zauważmy, że ostatnia postać nie jest postacią wykładniczą i różni się od postaci wyjściowej tym, że w celu zwiększenia czytelności zapisu grupy cyfr oddzielone są podkreślnikami. Należy pamiętać, że znak ten nie może przylegać do kropki dziesiętnej. Stosowanie znaku podkreślenia jako separatora grup cyfr jest również dopuszczalne w przypadku liczb całkowitych. Wolno więc pisać: 123_456_789E3 zamiast 123456789000.

W przypadku liczb rzeczywistych jest również możliwe wyrażanie tych liczb przy innych podstawach niż dziesiętna. Na przykład

$2\#111.01\# = 1 * 2^2 + 1 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} = 4 + 2 + 1 + 1/4 = 7.25$

Na koniec należy pamiętać, że liczby ujemne otrzymuje się przez działanie operatorem jednoargumentowym „−” na odpowiedni ciąg reprezentujący liczbę nieujemną. Na pytanie jakie wartości dziesiętne reprezentują ciągi:

16#E#E1, 2#11#E11, 16#F.FF#E+2, 2#1.1111_1111_111#E11

otrzymano następującą odpowiedź

224, 6144, 4.09500E+03, 4.09500E+03

wykonując następujący program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4  procedure Liczby_Dziesietne is
5  begin
6      Put (16#E#E1); New_Line;
7      Put (2#11#E11); New_Line;
8      Put (16#F.FF#E+2); New_Line;
9      Put (2#1.1111_1111_111#E11); New_Line;
10 end Liczby_Dziesietne;
```

2.3 Komentarze

Komentarze, czyli wyjaśnienia umieszczane są w treści programu w celu dostarczenia czytelnikowi dodatkowych informacji o strukturze i działaniu programu. Komentarze mogą opisywać np. rolę zmiennych i stałych, struktury danych, realizowane algorytmy, albo podawać źródła, w których można znaleźć ich kompletne opisy. Treść komentarzy opracowywana jest przez autora programu, przy czym autor powinien zawsze pamiętać, że treść programu może być czytana przez innych ludzi, a także przez samego autora, który po pewnym czasie zajmie się konserwacją swojego dzieła.

Mimo, że składnia Ady została tak pomyślana, aby zapewnić czytelność programów pisanych w tym języku, dobre komentarze są nieodzowne w celu uzyskania programów określanych jako samodokumentujące się. Nie jest oczywiste co należy uznać za „dobry komentarz", ale pewne istotne wskazówki można znaleźć np. w pracy (van Tassel 1982).

W Adzie komentarz jest dowolnym tekstem, który poprzedzony jest przylegającymi do siebie myślnikami (znakami minus). Końcem komentarza jest koniec linii, w której pojawił się znak komentarza „--". Oznacza to, że cały tekst napisany po tym znaku jest traktowany jako komentarz i jest ignorowany podczas kompilacji.

Jeżeli chcemy uzyskać dłuższy komentarz, to piszemy go w kilku liniach zaczynających się znakiem komentarza, co ilustruje następujący tekst:

```
-- Ten komentarz
-- piszemy
-- w trzech liniach
```

Na koniec warto dodać, że w komentarzach można używać znaków spoza zbioru znaków Latin-1. Wolno więc napisać:

```
--zmienna Dt_Fi reprezentuje pochodna ∂φ/∂t
```

Na koniec podajemy pewne zalecenia stylistyczne dotyczące komentarzy. Należy pisać programy z pełnymi objaśnieniami. Pierwszy komentarz powinien zostać umieszczony po nagłówku programu i powinien wyjaśniać do czego program służy. Można tam również podać algorytm wg którego realizowane są obliczenia oraz źródło, gdzie znajduje się opis algorytmu. Dodatkowo, pierwszy komentarz może obejmować dane autora programu i numer wersji, czy datę ostatniej modyfikacji programu. Van Tassel (1982) wymienia następujące zalecenia:

- ⇝ Stosuj komentarze wstępne (patrz wyżej)
- ⇝ Stosuj przewodniki po długich programach
- ⇝ Komentarz ma dawać coś więcej niż parafrazę teksu programu
- ⇝ Błędne komentarze są gorsze od zupełnego braku komentarzy

Parę zabawnych uwag na temat stylu pisania programów można znaleźć w (Green 2001)

2.4 Ćwiczenia

2

◁ Ćwiczenie 2.1 ▷▷▷ (Barnes 1998) Które z podanych niżej napisów nie są legalnymi identyfikatorami w Adzie i dlaczego?

⇝ Ada

⇝ ryba&frytki

⇝ szybkosc−przeplywu

⇝ UMO164G

⇝ Opoznienie__Czasowe

⇝ 77e2

⇝ X_

⇝ stopa opodatkowania

⇝ goto

◁ Ćwiczenie 2.2 ▷▷▷ (Barnes 1998) Które z podanych napisów nie są legalnymi zapisami liczb i dlaczego? Jakiego typu liczby oznaczają legalne zapisy?

⇝ 38.6

⇝ .5

⇝ 32e2

⇝ 32e−2

⇝ 2#1101

⇝ 2.71828_18285

⇝ 12#ABC#

⇝ E+6

⇝ 16#FfF#

⇝ 1_0#1_0#E1_0

⇝ 27.4e_2

⇝ 2#11#e−1

◁ Ćwiczenie 2.3 ▷▷▷ (Barnes 1998) Oblicz wartości reprezentowane przez liczby:

⇝ 16#E#E1

⇝ 2#11#E11

⇝ 16#F.FF#E+2

⇝ 2#1.1111_1111_1111#E11

◁ Ćwiczenie 2.4 ▷▷▷ (Barnes 1998) Na ile różnych sposobów można zapisać liczbę całkowitą 41? Nie należy brać pod uwagę podkreślników, rozróżnienia między E i e, nieznaczących zer na początku i opcjonalnego + przed wykładnikiem.

Rozdział 3

Skalarne typy danych

Przed omówieniem typów danych musimy zdefiniować kilka podstawowych pojęć:

Literałem nazywamy wartość danej jawnie pisaną w programie. Przykłady literałów można znaleźć na stronie 70.

Stałą nazywa się daną nie zmienianą w programie, do której odwołujemy sie przez identyfikator stałej.

Zmienną nazywa się obszar w pamięci, do którego odwołujemy się poprzez identyfikator. W miejscu tym przechowywana jest dana zmieniana (modyfikowana) podczas wykonywania programu.

3.1 Pojęcie typu danych

Zbiór dopuszczalnych wartości zmiennej wraz z podstawowymi operacjami określonymi w tym zbiorze nazywamy *typem zmiennej*. W języku Ada przyjęto zasadę, że każda zmienna musi mieć ściśle określony typ. Można wymienić kilka przyczyn ścisłego określania typów zmiennych.

Podanie zbioru dopuszczalnych wartości zmiennej wraz z identyfikatorem tej zmiennej (pod warunkiem, że nazwa tej zmiennej jest właściwie dobrana) określa rolę jaką zmienna pełni w programie, co znacznie zwiększa czytelność programu.

Mechanizm typizacji zmiennych zwiększa niezawodność oprogramowania, gdyż pozwala na wykrycie wielu błędów w programie tylko na podstawie analizy tekstu programu, np. próba podstawienia wartości logicznej do zmiennej liczbowej jest błędem wykrywanym przez kompilator. Innym przykładem błędu wykrywanego na poziomie kompilacji jest dodawanie zmiennej liczbowej do zmiennej typu logicznego.

Typizacja zmiennych umożliwia również wykrywanie błędów przekroczenia dopuszczalnego zakresu wartości zmiennych w trakcie wykonywania programu.

Jeżeli dla pewnej zmiennej określono, że jej wartości muszą należeć do przedziału 0..10 i zostanie jej nadana wartość 11, to powinno to spowodować sygnalizację błędu wykonania programu. Własność ta również zwiększa niezawodność oprogramowania.

Określenie typu zmiennej decyduje często o sposobie realizacji operacji wykonywanych na zmiennych tego typu. Dla przykładu: operacja dodawania zmiennych a, b jest inaczej realizowana w przypadku, gdy zmienne te są liczbami rzeczywistymi i inaczej, gdy są zmiennymi całkowitymi.

Kolejną przyczyną stosowania typizacji zmiennych jest określenie przez kompilator rozmiaru pamięci niezbędnego do reprezentacji odpowiedniej zmiennej.

Wymienione przyczyny spowodowały, że praktycznie we wszystkich nowoczesnych językach programowania, w tym również w Adzie, wprowadzono mechanizmy ścisłego deklarowania typów danych.[1]

Należy podkreślić, że stałe również mają swój typ, ale zbiór wartości każdej stałej jest jednoelementowy.

Ogólnie, typy danych można podzielić wg dwóch kryteriów. Pierwszy podział wyróżnia *typy zdefiniowane wstępnie* i *typy definiowane przez programistę*. Przez typy zdefiniowane wstępnie rozumiemy tutaj te typy danych, które określone są w module bibliotecznym[2] Standard i w związku z tym, nazywamy je dalej *typami standardowymi*. Pakiet ten jest dostępny w każdej implementacji Ady i zawiera określenia zbiorów wartości jakie mogą przyjmować dane typów standardowych oraz określenia podstawowych operacji, które można wykonywać na tych danych. Obiekty z tego pakietu i tylko z tego mogą być używane w programie bez konieczności jawnego wskazywania na liście importowej nazwy pakietu, z którego pochodzą. Innym kryterium podziału typów danych jest to, czy dana należąca do typu ma składowe, czy jest niepodzielna. Jeżeli typ tworzą niepodzielne elementy, to nazywamy go *typem elementarnym*[3], natomiast inne typy nazywamy *typami strukturalnymi*.

3.2 Definiowanie typów

Zazwyczaj, definicja czyli deklaracja typu przypisuje mu identyfikator, oraz określa zbiór wartości jakie mogą przyjmować dane należące do typu. Typami, które nie muszą otrzymywać nazw są tablice, zadania i obiekty chronione, przy czym w takim przypadku tracimy możliwość odwoływania się do typu przez jego identyfikator. Należy podkreślić, że każda deklaracja określa nowy typ, inny od istniejących i nie można wartości jednego typu przypisywać zmiennym innego typu. Wynika to z zasady ścisłej typizacji przyjętej przy projektowaniu języka Ada, który należy do grupy języków o tzw. silnej typizacji.

Najczęściej, deklaracja typu wygląda następująco:

```
type Nazwa_Typu is Opis_Typu;
```

[1] Warto dodać, że Ada jest pod tym względem bardziej restrykcyjna, niż inne języki programowania.

[2] Dalej, zgodnie z terminologią Ady moduły nazywane są pakietami

[3] Typy elementarne dzielimy na typy skalarne i typy wskaźnikowe omawiane w rozdziale 8

Słowa type i is są słowami kluczowymi, Nazwa_Typu jest identyfikatorem, natomiast Opis_Typu opisuje jaki zbiór wartości obejmuje deklarowany typ.

Praktycznie we wszystkich dalszych rozdziałach tego skryptu, Czytelnik napotka przykłady deklaracji typów.

Powstaje pytanie, czy typy Typ_I1 i Typ_I2 zdefiniowane poniżej są zgodne?

```
type Typ_I1 is new Integer;
type Typ_I2 is new Integer;
```


Podany zapis może sugerować, że deklaracje te zmieniają jedynie nazwę typu standardowego Integer i powinny być zgodne. Jest tak w wielu innych językach programowania, natomiast w Adzie te typy są różne. Uzasadnienie tego jest następujące: skoro programista chciał te typy odróżnić przez nadanie im innych nazw, to mają one inne znaczenie logiczne. Jeżeli zmodyfikujemy poprzednią deklarację i napiszemy

```
type Liczba_Słoni is new Integer;
type Roczne_Dochody is new Integer;
```

to natychmiast zrozumiemy dlaczego nowe typy Liczba_Słoni i Roczne_Dochody są logicznie różne i mają jedynie taką samą wewnętrzną reprezentację. Takie podejście do deklarowania typów zabezpiecza przed przypadkowym dodawaniem słoni do rocznych dochodów.

3.3 Podtypy

Przypuśćmy, że mamy dany pewien typ danych o nazwie Typ_Danych i chcemy ograniczyć zbiór wartości, które definiuje ten typ, oraz chcemy zachować możliwość stosowania wszystkich operacji określonych dla tego typu. W takiej sytuacji możemy skonstruować *podtyp* typu Typ_Danych, przy czym zbiór wartości podtypu definiuje się przez podanie *ograniczeń* w jego deklaracji. Typ na bazie, którego deklarujemy podtyp będzie tutaj nazywany *typem bazowym*, ale jak podaje Barnes (Barnes 1998) formalnie poprawniejsze jest stosowanie nazwy *pierwszy podtyp*. Przykładami deklaracji podtypów są następujące określenia:

```
subtype Numer_Dnia is Integer range 1..31;
subtype Numer_Miesiaca is Integer range 1..12;
subtype Punty_Kredytowe is Integer range 0..300;
subtype Odcinek_Jednostkowy is Float range 0..1.0;
```

Ogólnie deklaracja podtypu ma postać

```
subtype Podtyp is Typ_Skalarny range Dolne..Gorne;
```

W tej deklaracji Dolne i Gorne oznaczają wyrażenia wyznaczające ograniczenie dolne i górne jakie mogą przyjmować dane deklarowanego podtypu, przy czym wartości tych wyrażeń muszą być typu na bazie którego tworzymy deklarację i muszą być to *wyrażenia statyczne* tzn. takie, których wartości mogą być wyznaczone podczas kompilacji programu.

Podtyp nie jest nowym typem, ale raczej przeniesieniem operacji określonych dla typu na pewien podzbiór. Mamy tu pojęciową analogię ze strukturami matematycznymi, do których należy pojęcie przestrzeni wektorowej i jej podprzestrzeni (Trajdos 1998).

Możemy tworzyć wyrażenia, których część argumentów jest typu bazowego, a inne są jego podtypu. Mimo że operacje takie są poprawne, to należy liczyć się z możliwością wystąpienia błędów naruszenia ograniczeń. Dla ilustracji rozważmy następujący przykład:


```
Dzien : Dzien_Tygodnia;
Liczba : Integer;
...
Dzien := Liczba;
```

Przypisanie zmiennej Dzien wartości zmiennej Liczba jest całkowicie legalne, ale jeżeli Liczba ma wcześniej nadaną wartość poza zakresem podtypu Dzien_Tygodnia, to wystąpi wspomniany błąd.

3.4 Typy wyliczeniowe

Uproszczona definicja formalna typu wyliczeniowego (enumeration_type) jest następująca:

```
enumeration_type ::= type identifier is enumeration_type_definition
enumeration_type_definition ::= (identifier{, identifier}),
```

przy czym enumeration_type_definition nazywamy definicją typu wyliczeniowego. Z definicji tej wynika, że typ wyliczeniowy określa się pisząc słowo kluczowe type, po którym podajemy nazwę typu, czyli jego identyfikator, a następnie słowo kluczowe is i dalej wymieniamy, czyli wyliczamy elementy należące do określanego typu. Lista elementów typu wyliczeniowego ujęta jest w nawiasy okrągłe, a poszczególne elementy oddzielone są przecinkami. Ilość elementów nie jest ograniczona z góry, ale nie może być pusta, a więc zbiór wartości typu wyliczeniowego musi zawierać co najmniej jeden element. Oto przykłady typów wyliczeniowych:

```
type T is (A1, A2, A3, A4);
type Kolor is (Czerwony, Niebieski, Zielony, Fioletowy, Zloty);
type Dzien_Roboczy is (Poniedzialek, Wtorek, Sroda, Czwartek, Piatek);
type Dzien_Tygodnia is (Poniedzialek, Wtorek, Sroda, Czwartek,
                        Piatek, Sobota, Niedziela);
type Jedno_Elementowy is (Sam_Jeden);
```

Zauważmy, że mamy tu do czynienia z przykładem przeciążenia tzn. sytuacji, gdy te same nazwy oznaczają różne obiekty. Przy silnej typizacji jaka obowiązuje w Adzie typy Dzien_Roboczy i Dzien_Tygodnia są różnymi typami chociaż można odnieść wrażenie, że zawierają te same elementy. W celu uniknięcia niejednoznaczności możemy (a jeżeli kompilator może mieć wątpliwość o którą wartość chodzi – to nawet musimy) pisać:

```
Dzien_Roboczy'(Poniedzialek)
```

albo

```
Dzien_Tygodnia'(Poniedzialek).
```

Nie możemy jednak nazwy Poniedzialek użyć w tym samym momencie jako nazwy elementu typu wyliczeniowego i identyfikatora zmiennej. Deklaracja jednego obiektu przesłoni deklarację drugiego i obie deklaracje (typu i zmiennej) nie mogą pojawić się w tej samej części deklaracyjnej.

Na bazie typu wyliczeniowego można deklarować podtypy, nazywane czasami typami okrojonymi, przy czym określenie ograniczeń wyznaczających zbiór wartości należących do określanego podtypu ma postać:

```
range Dolne .. Gorne
```

Jak pamiętamy, Dolne i Gorne są wyrażeniami wyznaczającymi odpowiednio wartość dolną i górną przedziału, do którego należą dane określanego podtypu.

Zamiast więc deklarować typ Dzien_Roboczy wydaje się rozsądne zastosowanie deklaracji

```
subtype Dzien_Roboczy is Dzien_Tygodnia range Poniedzialek..Piatek;
D_Pracy: Dzien_Roboczy;
```

albo po prostu

```
D_Pracy: Dzien_Tygodnia range Poniedzialek .. Piatek;
```

W tym drugim przypadku tym zmiennej D_Pracy określony jest przez opis przedziału wartości jakie ta zmienna może przyjmować, a nie przez nazwę podtypu.

W sytuacji, gdy tworzymy podtyp na bazie pewnego typu wyliczeniowego, może się zdarzyć, że ograniczenie dolne jest większe od górnego. Jeżeli tak się zdarzy, to określimy podtyp pusty. Zauważmy jednak, że nie możemy utworzyć pustego podtypu typu Jedno_Elementowy.

Dane typu wyliczeniowego są uporządkowane zgodnie z kolejnością występowania identyfikatorów w liście określającej typ wyliczeniowy tzn. idąc od lewej do prawej. Do danych typów wyliczeniowych można stosować atrybuty ´First i ´Last, których zastosowanie umożliwia obliczenie pierwszego i ostatniego elementu danego typu. Mamy więc:

```
Kolor'First = Czerwony
Kolor'Last = Zloty
```

Poza tym, możemy obliczać poprzednik i następnik pewnego elementu. Do tego celu służą określone wstępnie atrybuty ´Pred i ´Succ, których zastosowanie dla elementu Niebieski daje wyniki:

```
Kolor'Pred (Niebieski)=Czerwony
Kolor'Succ (Niebieski)=Zielony
```

W ogólnym przypadku w nawiasie może być wyrażenie odpowiedniego typu wyliczeniowego. Trzeba przy tym pamiętać, że jeżeli chcemy obliczyć następnik ostatniego elementu, albo poprzednik elementu pierwszego, to zostanie zgłoszony wyjątek CONSTRAINT_ERROR.

Liczbę porządkową elementu typu wyliczeniowego oblicza się przy pomocy atrybutu ´Pos. Mamy np.

Kolor'Pos (Czerwony) = 0
Kolor'Pos (Niebieski) = 1
Kolor'Pos (Zielony) = 2

Zwróćmy uwagę, że liczba porządkowa pierwszego elementu jest równa 0, a nie 1.

Atrybutem odwrotnym do ´Pos jest ´Val. Mamy w związku z tym:

Kolor'Val (0) = Czerwony
Kolor'Pos (1) = Niebieski
Kolor'Pos (2) = Zielony

i ogólnie, jeżeli mamy pewien typ wyliczeniowy Typ_Wyl, to:

Typ_Wyl'Val(Liczba) = Element
Typ_Wyl'Val (Typ_Wyl'Pos(Element)) = Element

oraz

Typ_Wyl'Pos (Typ_Wyl'Val(Liczba))= Liczba

Ponadto jest jasne, że

Typ_Wyl'Succ(Element)= Typ_Wyl'Val (Typ_Wyl'Pos(Element)+ 1)

Wymienione atrybuty można stosować do podtypów określonych na bazie typu wyliczeniowego.

Ponieważ deklaracja typu wyliczeniowego narzuca porządek w zbiorze elementów tego typu, możemy stosować operatory relacyjne =, /=, <, <=, >,>=.

Piszemy więc

Czerwony < Niebieski
Poniedzialek >= Wtorek

Oczywiście w miejsce tych relacji można stosować porównanie wartości liczbowych określających pozycję elementów. Prowadzi to jednak do tego, że utożsamiamy elementy typu wyliczeniowego z liczbami odpowiadającymi ich pozycji, a bardziej naturalne jest używanie właściwych nazw elementów. Działanie podanych atrybuty typów wyliczeniowych ilustruje następujący program:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Atrybuty_Typu_Wyliczeniowego is
3     type Kolor is (Czerwony, Zielony, Niebieski, Fioletowy,
4                    Seledynowy);
5     subtype Kolor_Rgb is Kolor range Czerwony..Niebieski;
6  begin
7     Put ("Atrybuty typu Kolor i podtypu Kolor_RGB");
8     New_Line;
9     Put ("Pierwszy element = ");
10    Put (Kolor'Image(Kolor'First));
11    New_Line;
12    Put ("Ostatni element = ");
13    Put (Kolor'Image(Kolor'Last));
14    New_Line;
15    Put ("Nastepnik pierwszego = ");
16    Put (Kolor'Image(Kolor'Succ(Kolor'First)));
```

```
17      New_Line;
18      Put ("Poprzednik ostatniego = ");
19      Put (Kolor'Image(Kolor'Pred(Kolor'Last)));
20      New_Line;
21      Put ("Liczba porzadkowa pierwszego elementu = ");
22      Put (Integer'Image(Kolor'Pos(Kolor'First)));
23      New_Line;
24      Put ("Ilosc elementow typu = ");
25      Put (Integer'Image(Kolor'Pos(Kolor'Last) + 1));
26      New_Line;
27      Put ("Nastepnik ostatniego elementu podtypu Kolor_RGB = ");
28      Put (Kolor_Rgb'Image(Kolor_Rgb'Succ(Kolor_Rgb'Last)));
29      New_Line;
30  end Atrybuty_Typu_Wyliczeniowego;
```

Zgodnie z oczekiwaniem autora programu, otrzymano wyniki:

```
Atrybuty typu Kolor i podtypu Kolor_RGB
Pierwszy element = CZERWONY
Ostatni element = SELEDYNOWY
Nastepnik pierwszego = ZIELONY
Poprzednik ostatniego = FIOLETOWY
Liczba porzadkowa pierwszego elementu = 0
Ilosc elementow typu = 5
Nastepnik ostatniego elementu podtypu Kolor_RGB = FIOLETOWY
```

Pewne instrukcje, użyte w tym programie, wymagają omówienia. W szczególności są to instrukcje wywołania procedur Put i New_Line importowanych z pakietu Ada.Text_IO.

Zajmijmy się najpierw procedurą Put, która służy do wypisywania wartości napisów (typ String), czyli ciągów znaków (typ Character) na urządzeniu wyjściowym (w naszym przypadku jest nim domyślne urządzenie, czyli ekran monitora, a dokładniej okno, w którym obrazowane są wyniki działania programu). Podana w części definicyjnej pakietu Ada.Text_IO (plik a−textio.ads specyfikacja (nagłówek) procedury Put jest następująca:

```
procedure Put (Item : in String);
```

Informacje zawarte w tej specyfikacji umożliwiają prawidłowe wykorzystanie tej procedury, czyli poprawne jej wywołanie w naszym programie. Dla nas jest istotne to, że parametr formalny Item jest napisem (typ String) i stąd wynika, że parametr aktualny też musi być tego typu. W Adzie napisami są ciągi znaków (typ Character), przy czym typ standardowy String jest zdefiniowany jako jednowymiarowa tablica otwarta znaków, czyli ciąg o nieokreślonej ilości znaków. Oznacza to, że procedura może wypisać napis o dowolnej długości, ale przy wywołaniu musi być znana długość wypisywanego napisu. Przykładem jest instrukcja

```
Put ("Pierwszy element = ");
```

przy czym możemy sprawdzić, że wypisywany napis Pierwszy element =, ujęty w cudzysłowy, których nie wliczamy, ma 19 znaków.

Podane wyniki działania programu mogą sugerować, że procedura Put została wykorzystana do wypisywania wartości całkowitych. Przykładem jest instrukcja

```
Put(Integer'Image(Kolor'Pos(Kolor'Last) + 1));
```

Zauważmy jednak, że wartość wyrażenia Kolor'Pos(Kolor'Last)+ 1 jest liczbą całkowitą (typu Integer) i wartość ta jest zamieniana na odpowiedni napis wypisywany na ekranie przez zastosowanie atrybutu 'Image, który dla danego argumentu podtypu skalarnego wylicza odpowiadający temu argumentowi ciąg znaków (typ String), przy czym w tym przypadku tym podtypem jest typ Integer.

Podobnie wykorzystano procedurę Put do wypisania wartości typu wyliczeniowego Kolor, przy czym przykładem odpowiedniej instrukcji jest

```
Put (Kolor'Image(Kolor'Pred(Kolor'Last)));
```

Drugą procedurą importowaną z pakietu Ada.Text_IO jest New_Line, której nagłówek ma następującą postać:

```
procedure New_Line (Spacing : in Positive_Count := 1);
```

przy czym podtyp Positive_Count obejmuje liczby całkowite z zakresu 1..Natural'Last, czyli wszystkie dodatnie liczby całkowite dostępne w danej implementacji (podtyp Natural omawiamy w następnym podrozdziale).

Zauważmy, że w nagłówku procedury New_Line zdefiniowano parametr formalny Spacing i nadano mu tam wartość domyślną 1. Oznacza to, że jeżeli przy wywołaniu tej procedury chcemy użyć takiej samej wartości parametru aktualnego, to możemy wywołać tę procedurę z pustą listą parametrów aktualnych, czyli piszemy New_Line. Efektem takiego wywołania jest pisanie następnych komunikatów od początku następnej linii. Jeżeli chcemy pisać w linii o 1 niższej, to piszemy New_Line(2) i ogólnie New_Line(Ilosc_Linii), pod warunkiem, że Ilosc_Linii jest z zakresu 1..Natural'Last.

Jak wspomniano, w Adzie istnieje zdefiniowany wstępnie moduł biblioteczny, czyli pakiet o nazwie Standard, w którym określono m.in. następujące typy, nazywane przez nas typami standardowymi: Boolean, Integer, Float, Character, i String. W następnych kilku punktach omówimy kolejno cztery pierwsze typy, natomiast typ String omawiamy w rozdziale dotyczącym typów strukturalnych.

3.5 Ćwiczenia

◁ Ćwiczenie 3.1 ▷▷▷ (Barnes 1998) Dana jest deklaracja

```
type Pan_Mlody is (Druciarz, Krawiec, Zolnierz, Marynarz, Bogacz,
    Biedak, Zebrak, Zlodziej);
```

Mamy ośmiu kandydatów na męża pewnej damy reprezentowanych przez dane typu Pan_Mlody i każdy z nich zjada kawałek dużego tortu. Napisać wyrażenie, które określi pana młodego, który zjadł N-ty kawałek tortu. Obliczyć wynik dla N = 10.

3.6 Typ standardowy `Boolean`

Typ logiczny, nazywany typem Boolean jest zdefiniowanym wstępnie typem wyliczeniowym, którego definicja w pakiecie Standard jest następująca:

```
type Boolean is (False, True);
```

Wartość False oznacza fałsz, czyli zero logiczne, a True oznacza prawdę, czyli jedynkę logiczną. Zero i jeden odpowiadają porządkowi w jakim umieszczono odpowiednie nazwy w liście określającej omawiany typ. Można się o tym przekonać kompilując i wykonując program:


```
1  with Ada.Text_IO; use Ada.Text_Io;
2  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3  procedure Liczby_Porzadkowe_Boolean is
4  begin
5     Put ("Wartosc odpowiadajaca False = ");
6     Put (Boolean'Pos(False), 4);
7     New_Line;
8     Put ("Wartosc odpowiadajaca True =  ");
9     Put (Boolean'Pos(True), 4);
10 end Liczby_Porzadkowe_Boolean;
```

Bardzo często w programach komputerowych kolejność wykonywania instrukcji wynika z tego czy pewien warunek jest prawdziwy, czy nieprawdziwy. Typowym, chociaż bardzo prostym przykładem jest nasz pierwszy program służący do obliczania pierwiastków trójmianu kwadratowego. W programie tym najpierw obliczaliśmy wartość wyróżnika W i w zależności od wartości relacji $W < 0.0$ obliczaliśmy pierwiastki zespolone, albo rzeczywiste. Warunki tego rodzaju są w ogólnym przypadku wyrażeniami z argumentami typu Boolean. Do konstrukcji takich wyrażeń, nazywanych czasami *wyrażeniami logicznymi* mamy określone wstępnie *operatory logiczne*: jednoargumentowy not i dwuargumentowe and, or i xor. Operatory te nazywane są odpowiednio *operatorem negacji*, *iloczynu logicznego*, czyli *koniunkcji*, *sumy logicznej* czyli *alternatywy* i *sumy wykluczającej*. Sposób działania tych operatorów jest następujący:

Negacja

Argument	not Argument
True	False
False	True

Koniunkcja

Lewy	Prawy	Lewy and Prawy
False	False	False
False	True	False
True	False	False
True	True	True

Alternatywa

Lewy	Prawy	Lewy or Prawy
False	False	False
False	True	True
True	False	True
True	True	True

Alternatywa wykluczająca

Lewy	Prawy	Lewy xor Prawy
False	False	False
False	True	True
True	False	True
True	True	False

Przy tworzeniu wyrażeń logicznych należy brać pod uwagę priorytety operatorów logicznych. Priorytet operatora not jest wyższy od operatorów and, or i xor. Te operatory dwuargumentowe mają równy priorytet[4], przy czym ich priorytet jest niższy od priorytetów wszystkich innych operatorów, a w szczególności od operatorów relacyjnych =, /=, <, <=, >, >=. Z tego wynika, że możemy pisać:

A < B and I = J,

jednak nie wolno napisać:

A and B or C

W celu uzyskania prawidłowego wyrażenia należy użyć nawiasów:

(A and B)or C albo A and (B or C)

dzięki czemu zmniejsza się niebezpieczeństwo popełnienia trudnych do wykrycia błędów, tak jak ma to miejsce w niektórych innych językach programowania.

Jeżeli stosujemy kilka razy ten sam operator, to nie musimy stosować nawiasów. Można więc napisać:

A or B or C, albo A xor B xor C

przy czym wartościowanie wyrażenia odbywa się od lewej do prawej.

Ponieważ operator negacji ma wyższy priorytet od pozostałych, zdefiniowanych wstępnie operatorów logicznych, musimy umiejętnie stosować nawiasy. Tak więc:

not A and B oznacza (not A)and B, a nie not (A and B).

Tę własność ilustruje poniższy program służący do wypisywania wartości funkcji dwuargumentowych Nand i Nor, które odgrywają podstawową rolę w konstrukcji sprzętu komputerowego. Wynika to z tego, że dowolna funkcja boolowska może być skonstruowana przy pomocy operacji Nand, albo Nor (Arbib, Kfoury i Moll 1981, Birkhoff i Bartee 1983). Układy elektroniczne realizujące te funkcje nazywane są odpowiednio bramkami Nand i Nor, natomiast funkcje realizowane przez te bramki nazywane są czasami kreską Sheffera i strzałką Pierce'a.

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Nand_Nor is
3
4     function Nand (X, Y : Boolean ) return Boolean is
5     begin
6        return not (X and Y);
7     end Nand;
8
9     function Nor (X, Y : Boolean ) return Boolean is
10    begin
```

[4] Inaczej niż np. w C, czy w Javie!

```
11       return not (X or Y);
12    end Nor;
13
14 begin
15    Put_Line ("Nand = not(and)");
16    for Lewy in False .. True loop
17       for Prawy in False .. True loop
18          New_Line;
19          Put ("Lewy = ");
20          Put (Boolean'Image(Lewy));
21          Set_Col (20);
22          Put ("Prawy = ");
23          Put (Boolean'Image(Prawy));
24          Set_Col (40);
25          Put ("Nand (Lewy, Prawy) = ");
26          Put (Boolean'Image(Nand (Lewy, Prawy)));
27       end loop;
28    end loop;
29    New_Line (2);
30    Put_Line ("Nor = not(or)");
31    for Lewy in False .. True loop
32       for Prawy in False .. True loop
33          New_Line;
34          Put ("Lewy = ");
35          Put (Boolean'Image(Lewy));
36          Set_Col (20);
37          Put ("Prawy = ");
38          Put (Boolean'Image(Prawy));
39          Set_Col (40);
40          Put ("Nor (Lewy, Prawy) = ");
41          Put (Boolean'Image(Nor(Lewy, Prawy)));
42       end loop;
43    end loop;
44 end Nand_Nor;
```

Wykonanie tego programu dało następujące wyniki:

```
Nand = not(and)

Lewy = FALSE       Prawy = FALSE       Nand (Lewy, Prawy) = TRUE
Lewy = FALSE       Prawy = TRUE        Nand (Lewy, Prawy) = TRUE
Lewy = TRUE        Prawy = FALSE       Nand (Lewy, Prawy) = TRUE
Lewy = TRUE        Prawy = TRUE        Nand (Lewy, Prawy) = FALSE

Nor = not(or)

Lewy = FALSE       Prawy = FALSE       Nor (Lewy, Prawy) = TRUE
Lewy = FALSE       Prawy = TRUE        Nor (Lewy, Prawy) = FALSE
Lewy = TRUE        Prawy = FALSE       Nor (Lewy, Prawy) = FALSE
Lewy = TRUE        Prawy = TRUE        Nor (Lewy, Prawy) = FALSE
```

W programie tym do sformatowania wyników wykorzystano dwie dotychczas nieznane procedury: Put_Line i Set_Col importowane z pakietu Ada.Text_IO. Nagłówki tych procedur podano niżej, przy czym poprzedzono je deklaracjami

typów, do których odwołują się definicje parametrów formalnych procedury Set_Col i omawianej w poprzednim podrozdziale procedury New_Line.

```
1 type Count is range 0 .. Natural'Last;
2 subtype Positive_Count is Count range 1 .. Count'Last;
3 procedure Put_Line (Item : in String);
4 procedure Set_Col (To : in Positive_Count);
```


Wywołanie procedury Put_Line powoduje zobrazowanie napisu, który jest jej parametrem aktualnym i przejście do nowej linii. Jest to równoważne do wywołania po sobie procedur Put i New_Line. Jeżeli chcemy wypisywać wartości napisów zaczynając od wybranej kolumny, to możemy wywołanie procedury Put poprzedzić wywołaniem procedury Set_Col z parametrem aktualnym równym numerowi kolumny, od której chcemy rozpocząć pisanie.

Poza tym, w programie Nand_Nor wykorzystano pewne konstrukcje, które nie są jeszcze znane i zainteresowany Czytelnik może znaleźć odpowiednie wyjaśnienia w podrozdziałach omawiających instrukcje pętli oraz funkcje.

Uzupełnienie

Oprócz wymienionych, istnieją jeszcze operacje, które zaliczamy tu do operacji logicznych. Są to or else, and then, in oraz not in. Pierwsze dwie odpowiadają alternatywie i koniunkcji i ich użycie ma postać:

Lewy or else Prawy, Lewy and then Prawy

W przypadku operatorów or i and[5] obliczane są obydwa argumenty, przy czym kolejność wartościowania tych argumentów nie jest określona. W sytuacji, gdy stosujemy operatory or else, and then zawsze obliczana jest wartość lewego argumentu, a wartość prawego obliczana jest tylko wtedy, gdy jest potrzebna do jednoznacznego wyznaczenia wyniku operacji. W związku z tym, jeżeli obliczamy warunek Lewy or else Prawy i Lewy = True, to niezależnie od wartości Prawy cały warunek będzie prawdziwy. Analogicznie jest w przypadku obliczania Lewy and then Prawy. Jeżeli Lewy = False, to wiadomo, że całe wyrażenie przyjmie wartość False. Powstaje naturalne pytanie kiedy stosować or, albo and, a kiedy ich odpowiedniki or else i and then. Zaleca się stosowanie tych drugich, gdy kolejność obliczania argumentów ma znaczenie. Dla przykładu przypuśćmy, że obliczamy warunek $A/B < Delta$ i chcemy zabezpieczyć się przed dzieleniem przez 0.0. Możemy w takiej sytuacji napisać:

if B /= 0 and then A/B < Delta then

co zapewni pożądane bezpieczeństwo.

Dodajmy, że priorytet tych operatorów jest taki sam jak pozostałych dwuargumentowych operatorów logicznych.

Pozostałe dwa operatory służą do sprawdzenia czy pewna dana typu skalarnego należy, albo nie należy do pewnego przedziału wartości. Możemy pisać:

Liczba not in 1..10;
Dim in Positive;
Dzisiaj in Dzien_Roboczy;

[5] Trzeba pamiętać, że Lewy i Prawy mogą być wyrażeniami, których argumentami mogą być wywołania funkcji logicznych.

albo

```
Dzisiaj in Poniedzialek..Piatek
```

Nie możemy jednak napisać:

```
Dzisiaj in Dzien_Tygodnia range Poniedzialek..Piatek
```

Operacja not in jest równoważna do zastosowania operatora in i negacji wyniku. Pierwsze z podanych wyrażeń jest więc równoważne do następującego:

```
not (Liczba in 1..10)
```

przy czym zwracamy uwagę, że musieliśmy użyć nawiasów, ponieważ priorytet operatów in oraz not in jest taki sam jak zwykłych operatorów relacyjnych.

3.7 Typ standardowy `Integer`

Typ Integer reprezentuje liczby całkowite, przy czym, jak już wspomniano, liczby całkowite należące do zbioru określonego przez ten typ reprezentowane są w sposób dokładny. Jaki podzbiór zbioru liczb całkowitych obejmuje typ Integer zależy od implementacji języka. Najmniejsza i największa liczba całkowita należąca do tego typu, a także ilość bitów prze-znaczona na daną tego typu może być obliczona przy pomocy atrybutów ´First, ´Last i ´Size działających na tym typie, co ilustruje poniższy, krótki program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Atrybuty_Integer is
4  begin
5      Put ("Atrybuty typu Integer ");  New_Line;
6      Put ("Najmniejsza liczba calkowita = ");
7      Put (Integer'Image(Integer'First));  New_Line;
8      Put ("Najwieksza liczba calkowia = ");
9      Put (Integer'Image(Integer'Last));  New_Line;
10     Put ("Wielkosc liczby calkowitej w bitach = ");
11     Put (Integer'Image(Integer'Size));  New_Line;
12 end Atrybuty_Integer;
```

Jeżeli skompilujemy ten program przy pomocy systemu kompilacyjnego Ada-Gide v. 6.22.10 i wykonamy na komputerze z procesorem Pentium i systemem operacyjnym Microsoft Windows, to otrzymamy wynik:

```
Atrybuty typu Integer
Najmniejsza liczba calkowita = -2147483648
Najwieksza liczba calkowia = 2147483647
Wielkosc liczby calkowitej w bitach = 32
```

Oprócz tego typu mogą występować inne typy całkowite zdefiniowane wstępnie. Mogą to być Short_Integer i Long_Integer. Wykonanie programu Atrybuty_Integer po wymianie Integer na Short_Integer daje wynik:

```
Atrybuty typu Short_Integer
Najmniejsza liczba calkowita = -32768
Najwieksza liczba calkowia = 32767
Wielkosc liczby calkowitej w bitach = 16
```

natomiast po wymianie Integer na Long_Integer otrzymano:

```
Atrybuty typu Long_Integer
Najmniejsza liczba calkowita = -2147483648
Najwieksza liczba calkowia = 2147483647
Wielkosc liczby calkowitej w bitach = 32
```


Na danych typu Integer można wykonywać znane z matematyki (Trajdos 1998) operacje algebraiczne i obliczać wartości relacji, jedno i dwuargumentowych. Do tego celu służą *operatory*, które działają na argumenty omawianego typu. Wyróżniamy trzy grupy operatorów: *relacyjne, algebraiczne jednoargumentowe (unarne) i algebraiczne dwuargumentowe (binarne).*

Operatory relacyjne używane są do obliczania wartości relacji porównania dwóch liczb całkowitych, a więc obliczają wartość logiczną, czyli typu Boolean. Listę tych operatorów zawiera tabela 3.1.

Tabela 3.1: Operatory relacyjne określone dla argumentów całkowitych

Nazwa operatora	**Oznaczenie**	**Rodzaj**	**Typy argumentów**	**Typ wyniku**
Równość	=	binarny	Integer, Integer	Boolean
Nierówność	/=	binarny	Integer, Integer	Boolean
Mniejszość	<	binarny	Integer, Integer	Boolean
Mniejszość lub równość	<=	binarny	Integer, Integer	Boolean
Większość	>	binarny	Integer, Integer	Boolean
Większość lub równość	>=	binarny	Integer, Integer	Boolean

Zdefiniowane wstępnie operatory algebraiczne podano w tabeli 3.2.

Tabela 3.2: Operatory algebraiczne określone dla argumentów całkowitych

Nazwa operatora	**Oznaczenie**	**Rodzaj**	**Typy argumentów**	**Typ wyniku**
Identyczność	+	unarny	Integer	Integer
Zmiana znaku	-	unarny	Integer	Integer
Wartość bezwzględna	abs	unarny	Integer	Integer
Dodawanie	+	binarny	Integer, Integer	Integer
Odejmowanie	-	binarny	Integer, Integer	Integer
Mnożenie	*	binarny	Integer, Integer	Integer
Dzielenie	/	binarny	Integer, Integer	Integer
Reszta z dzielenia	rem	binarny	Integer, Integer	Integer
Reszta z dzielenia	mod	binarny	Integer, Integer	Integer
Potęgowanie	**	binarny	Integer, Natural	Integer

Własności operatorów standardowych działających na liczbach całkowitych ilustrują poniższe wyniki:

```
-- Operatory relacyjne.
Argument lewy = -7, Argument prawy = 5
(Lewy = Prawy) = False
(Lewy /= Prawy) = True
(Lewy < Prawy) = True
(Lewy <= Prawy) = True
(Lewy > Prawy) = False
(Lewy >= Prawy) = False
-- Operatory algebraiczne unarne.
Argument = -100
  + (Argument) = -100
  - (Argument) = 100
abs (Argument) = 100
-- Operatory algebraiczne binarne.
Argument lewy = -7, Argument prawy = 5
Argument = -100, Wykladnik = 2
(Lewy + Prawy) = -2
(Lewy - Prawy) = -12
(Lewy * Prawy) = -35
(Lewy / Prawy) = -1
(Lewy mod Prawy) = 3
(Lewy rem Prawy) = -2
(Argument ** Wykladnik) = 10000
```

Należy zwrócić uwagę na to, że znaki + i − oznaczają dwa rodzaje operatorów: jednoargumentowe i dwuargumentowe. Używanie tych samych oznaczeń do różnych operacji nazywamy przeciążaniem operatorów. Stosowanie + jako operatora jednoargumentowego jest formalnie poprawne, ale nie zmienia wartości wyrażenia. Trzeba jednak pamiętać, że nie należy pisać:

$$A + +B,\ A - -B,\ A + -B.$$

Poprawne formy tych wyrażeń otrzymamy pisząc:

$$A + (+B),\ A - (-B),\ A + (-B).$$

chociaż prościej jest:

$$A + B,\ A + B,\ A - B.$$

Zwróćmy uwagę na to, że wyeliminowaliśmy w ten sposób działania jednoargumentowe.

Należy teraz omówić różnicę pomiędzy operatorami rem i mod.

Przy dzieleniu dwóch liczb całkowitych otrzymuje się iloraz i resztę, które są liczbami całkowitymi. Jeżeli A oznacza dzielną, B oznacza dzielnik, Q oznacza iloraz i R oznacza resztę, to liczby te spełniają równanie

$$A = Q * B + R$$

W Adzie mamy operator dwuargumentowy / taki, że

$$Q = A/B$$

oraz operator dwuargumentowy rem taki, że

$$R = A \operatorname{rem} B$$

Należy zwrócić uwagę na to, że jeżeli A i B nie dzielą się bez reszty, to iloraz Q jest zawsze liczbą bliższą zeru. Mamy więc

$$\begin{array}{cc} 5/2 = 2 & 5/(-2) = -2 \\ (-5)/2 = -2 & (-5)/(-2) = 2 \end{array}$$

Z tego widzimy, że wartość bezwzględna ilorazu jest taka sama jak iloraz wartości bezwzględnych argumentów. Zgodnie z podanym równaniem, operator obliczający resztę działa następująco:

$$\begin{array}{cc} 5 \operatorname{rem} 2 = 1 & 5 \operatorname{rem} (-2) = 1 \\ (-5) \operatorname{rem} 2 = -1 & (-5) \operatorname{rem} (-2) = -1 \end{array}$$

Drugim operatorem obliczającym resztę jest operator mod. Jest to również operator dwuargumentowy wewnętrzny, a więc wartość operacji A mod B jest liczbą całkowitą. W przypadku, gdy dzielnik jest dodatni A mod B należy do przedziału $[0..B-1]$, natomiast gdy dzielnik jest ujemny A mod B należy do przedziału $[B+1..0]$. Mamy więc:

$$\begin{array}{rcl} 7 \bmod 3 & = & 1 \\ (-7) \bmod 3 & = & 2 \\ 7 \bmod (-3) & = & -2 \\ (-7) \bmod (-3) & = & -1 \end{array}$$

Zauważmy, że w przypadku operatora mod znak wyniku określony jest przez znak drugiego argumentu - dzielnika, natomiast w przypadku operatora rem znak ten określony jest przez znak pierwszego argumentu - dzielnej.

Możemy zinterpretować operator mod jako resztę z dzielenia całkowitego, gdy iloraz jest obliczany przy pomocy funkcji nazywanej częścią całkowitą - Entier. Funkcja ta jest często oznaczana symbolem

$$\lfloor \cdot \rfloor$$

i określona jest następująco (Ross i Wright 1999):

$$\lfloor x \rfloor =$$

największej liczbie całkowitej takiej, że jest mniejsza lub równa x.

Mamy więc

$$\lfloor 4.5 \rfloor = 4, \ \lfloor -4.5 \rfloor = 5.$$

Jeżeli mamy

$$A = \tilde{Q} * B + \tilde{R},$$

to

$$\frac{A}{B} = \tilde{Q} + \frac{\tilde{R}}{B},\ 0 \leqslant \frac{\tilde{R}}{B} < 1.$$

W tym przypadku

$$\tilde{Q}$$

jest częścią całkowitą liczby

$$A/B,$$

a więc

$$\tilde{Q} = \lfloor A/B \rfloor .$$

Nietrudno zauważyć, że jeżeli znaki argumentów są różne, to wyniki operacji

$$A \operatorname{rem} B$$

i

$$A \bmod B$$

różnią się. Warto na koniec dodać, że operacja mod służy do implementacji arytmetyki modulo. Dla przykładu mamy

$$(A + B) \bmod N = (A \bmod N + B \bmod N) \bmod N$$

Operacja potęgowania liczb całkowitych nie jest typową operacją algebraiczną dwuargumentową, ponieważ drugi argument, czyli wykładnik musi być liczbą całkowitą nieujemną. W Adzie zbiór tych liczb reprezentowany jest przez zdefiniowany wstępnie podtyp Natural, którego określenie jest następujące:

```
subtype Natural is Integer range 0..Integer'Last;
```

Drugim podtypem całkowitym zdefiniowanym wstępnie jest

```
subtype Positive is Integer range 1..Integer'Last;
```

Jak wiadomo operatory są używane do tworzenia wyrażeń. Sposób obliczania wartości wyrażenia, a więc sposób *wartościowania wyrażenia* określony jest przez hierarchię operatorów, czyli *priorytety operatorów*.

Priorytety operatorów działających na argumentach całkowitych przedstawiono poniżej w porządku od najwyższego do najniższego.

**	abs					
*	/	mod	rem			
+	−					jednoargumentowe (unarne)
+	−					dwuargumentowe (binarne)
=	/=	<	<=	>	>=	

Jeżeli mamy wyrażenie, w którym występuje kilka operatorów o tym samym priorytecie, to wartościowanie odbywa się w kolejności od lewej do prawej. Jeżeli chcemy narzucić sposób wartościowania, to możemy stosować nawiasy okrągłe, podobnie jak w algebrze elementarnej. Mamy więc

```
    A / B * C   ⇔   (A / B)* C
A + B * C + D   ⇔   A + (B * C)+ D
A * B + C * D   ⇔   (A * B)+ (C * D)
    A * B ** C  ⇔   A * (B ** C)
A * abs B * C   ⇔   A * (abs B)* C
```


W przypadku potęgowania nie wolno jednak pisać

A ** B ** C

i musimy, w zależności od naszych potrzeb napisać

(A ** B)** C lub A ** (B ** C)

Prawidłowe są jednak wyrażenia

A + B + C, A * B * C, A / B / C.

Zwróćmy uwagę, że poprawne jest wyrażenie

− A ** B ⇔ − (A ** B)

natomiast niepoprawny jest zapis A ** −B, który należy zastąpić przez A ** (−B).

Podobnie nie należy używać abs −A i trzeba zastosować nawiasy tzn. abs (−A).

Reasumując, radzimy używać nawiasów, nawet wtedy gdy nie jest to konieczne, ponieważ zwiększamy dzięki temu czytelność programu oraz unikamy błędów niewykrywalnych przez kompilator. Cytujemy tu Van Tassela (van Tassel 1982):

NAWIASY KOSZTUJĄ MNIEJ NIŻ BŁĘDY

3.8 Typ standardowy `Float`

Zajmijmy się teraz typem standardowym Float, służącym do reprezentacji liczb rzeczywistych. Podobnie jak w przypadku typu Integer, definicja typu Float zależy od implementacji języka. Z jaką, konkretną implementacją mamy do czynienia możemy sprawdzić kompilując i wykonując na naszym komputerze następujący program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Atrybuty_Float is
4  begin
5     Put ("Atrybuty liczb zmiennoprzecinkowych typu Float");
6     New_Line;
7     Put ("Najmniejsza liczba typu Float  = ");
8     Put (Float'Image(Float'First));
9     New_Line;
10    Put ("Najwieksza liczba typu Float  = ");
11    Put (Float'Image(Float'Last));
```

```
12    New_Line;
13    Put ("Wielkosc liczby typu Float w bitach = ");
14    Put (Integer'Image(Float'Size));
15 end Atrybuty_Float;
```

Program ten skompilowany w systemie AdaGide v. 6.22.10 na komputerze klasy PC zarządzanym przez system operacyjny Microsoft Windows dał następujące wyniki:

```
Najmniejsza liczba typu Float = −3.40282E+38
Najwieksza liczba typu Float = 3.40282E+38
Wielkosc liczby typu Float w bitach = 32
```


W naszym systemie mamy również określone wstępnie typy Short_Float, Long_Float i Long_Long_Float, a sposób reprezentacji i zakresy liczb rzeczywistych reprezentowanych tymi typami otrzymuje się modyfikując odpowiednio nasz program. Daje to następujące wyniki:

```
Najmniejsza liczba typu Short_Float = −3.40282E+38
Najwieksza liczba typu Short_Float = 3.40282E+38
Wielkosc liczby typu Short_Float w bitach = 32
Najmniejsza liczba typu Long_Float = −1.79769313486232E+308
Najwieksza liczba typu Long_Float = 1.79769313486232E+308
Wielkosc liczby typu Long_Float w bitach = 64
Najmniejsza liczba typu Long_Long_Float = −1.18973149535723177E+4932
Najwieksza liczba typu Long_Long_Float = 1.18973149535723177E+4932
Wielkosc liczby typu Long_Long_Float w bitach = 96
```

Dalej zadowolimy się typem Float, który jest wystarczający dla potrzeb naszego, wstępnego kursu programowania.

Tabela 3.3: Operatory relacyjne określone dla argumentów rzeczywistych

Nazwa operatora	**Oznaczenie**	**Rodzaj**	**Typy argumentów**	**Typ wyniku**
Równość	=	binarny	Float, Float	Boolean
Nierówność	/=	binarny	Float, Float	Boolean
Mniejszość	<	binarny	Float, Float	Boolean
Mniejszość lub równość	<=	binarny	Float, Float	Boolean
Większość	>	binarny	Float, Float	Boolean
Większość lub równość	>=	binarny	Float, Float	Boolean

Na liczbach rzeczywistych można wykonywać operacje algebraiczne oraz wyznaczać wartości relacji. Do tych celów służą operatory relacyjne oraz operatory algebraiczne jedno- i dwuargumentowe podane w tabelach 3.3 i 3.4.

Zasady wartościowania wyrażeń, w których występują składniki i czynniki typu Float są takie same jak w przypadku odpowiednich wyrażeń z argumentami całkowitymi. Warto jednak zwrócić uwagę (patrz Tabela 3.4), że w przypadku potęgowania liczb rzeczywistych, wykładnik może być liczbą całkowitą ujemną. Własności operatorów standardowych działających na liczbach rzeczywistych ilustruje poniższy przykład:

Tabela 3.4: Operatory algebraiczne określone dla argumentów rzeczywitych

Nazwa operatora	Oznaczenie	Rodzaj	Typy argumentów	Typ wyniku
Identyczność	+	unarny	Float	Float
Zmiana znaku	−	unarny	Float	Float
Wartość bezwzględna	abs	unarny	Float	Float
Dodawanie	+	binarny	Float, Float	Float
Odejmowanie	−	binarny	Float, Float	Float
Mnożenie	*	binarny	Float, Float	Float
Dzielenie	/	binarny	Float, Float	Float
Potęgowanie	**	binarny	Float, Integer	Float

```
− Operatory relacyjne.
Argument lewy = −2.50, Argument prawy = 10.00
(Lewy = Prawy) = False
(Lewy /= Prawy) = True
(Lewy < Prawy) = True
(Lewy <= Prawy) = True
(Lewy > Prawy) = False
(Lewy >= Prawy) = False
− Operatory algebraiczne unarne.
Argument = −5.00
  + (Argument) = −5.00
  − (Argument) = 5.00
abs (Argument) = 5.00
− Operatory algebraiczne binarne.
Argument lewy = −2.50, Argument prawy = 10.00
Argument = −5.00, Wykladnik = −2
(Lewy + Prawy) = 7.50
(Lewy − Prawy) = −12.50
(Lewy * Prawy) = −25.00
(Lewy / Prawy) = −0.25
(Argument ** Wykladnik) = 0.04
```

Do danych typów zmiennopozycyjnych można stosować atrybuty 'First, 'Last, 'Size, 'Digits, 'Model_Epsilon, 'Safe_First, 'Safe_Last. Sposób działania trzech pierwszych już znamy, natomiast działanie czterech ostatnich ilustruje program:

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3  with Ada.Float_Text_IO; use Ada.Float_Text_IO;
4
5  procedure Atrybuty_Rzeczywiste is
6  begin
7     Put ("Nastepna reprezentowalna po 1.0 = ");
8     Put (1.0 + Float'Model_Epsilon, 2, 10);
9     New_Line;
10    Put ("Reprezentacja liczby 1.0        = ");
11    Put (1.0, 2, 10);
12    New_Line;
```

```
13    Put ("Nastepna reprezentowalna - 1.0 = ");
14    Put (Float'Model_Epsilon, 2, 10);
15    New_Line;
16    Put ("Ograniczenie dolne              = ");
17    Put (Float'Safe_First, 2, 10);
18    New_Line;
19    Put ("Ograniczenie gorne              = ");
20    Put (Float'Safe_Last, 2, 10);
21    New_Line;
22    Put ("Ilosc cyfr w reprezentacji");
23    New_Line;
24    Put ("typ Float                       = ");
25    Put (Float'Digits, 2);
26    New_Line;
27    Put ("typ Long_Float                  = ");
28    Put (Long_Float'Digits, 2);
29    New_Line;
30    Put ("typ Long_Long_Float             = ");
31    Put (Long_Long_Float'Digits, 2);
32 end Atrybuty_Rzeczywiste;
```

Program ten wyprodukował następujące wyniki:

```
Nastepna reprezentowalna po 1.0 = 1.0000001192E+00
Reprezentacja liczby 1.0 = 1.0000000000E+00
Nastepna reprezentowalna - 1.0 = 1.1920928955E-07
Ograniczenie dolne = -3.4028234664E+38
Ograniczenie gorne = 3.4028234664E+38
Ilosc cyfr w reprezentacji
typ Float = 6
typ Long_Float = 15
typ Long_Long_Float = 18
```

W podanym programie do wypisywania wartości typów Float i Integer zastosowano procedury Put importowane odpowiednio z pakietów Ada.Integer_Text_IO i Ada.Float_Text_IO. Pakiety te są konkretyzacjami pakietów ogólnych Ada.Text_IO.Integer_IO i Ada.Text_IO.Float_IO odpowiednio dla typów Integer i Float i zawierają podprogramy do obsługi operacji wejścia i wyjścia wykonywanych na danych tych typów liczbowych. Poza wspomnianymi podprogramami pakiety zawierają potrzebne deklaracje typów, podtypów, zmiennych i stałych. W pakiecie definicyjnym Ada.Text_IO.Integer_IO znajdują się m.in. następujące definicje:

```
1 type Num is range <>;
2
3 Default_Width : Field := Num'Width;
4 Default_Base : Number_Base := 10;
5
6 procedure Put ( Item : in Num;
7                 Width : in Field := Default_Width;
8                 Base : in Number_Base := Default_Base);
```

przy czym Field jest podtypem importowanym z pakietu Ada.Text_IO, gdzie znajdziemy deklarację postaci

```
subtype Field is Integer range 0 .. 255;
```

Z kolei w pakiecie definicyjnym Ada.Text_IO.Float_IO podane są deklaracje stosowane w przypadku typów zmiennoprzecinkowych

```
type Num is digits <>;

Default_Fore : Field := 2;
Default_Aft : Field := Num'Digits - 1;
Default_Exp : Field := 3;

procedure Put ( Item : in Num;
                Fore : in Field := Default_Fore;
                Aft : in Field := Default_Aft;
                Exp : in Field := Default_Exp);
```

3.9 Typ standardowy Character

Znaki w Adzie określone są wstępnie przez typ Character reprezentujący zbiór znaków kodu ISO 8859-1, nazywany czasami kodem Latin-1 (wykaz znaków zdefiniowanych tym kodem można znaleźć w tabeli 5.1). Jest to kod 8-bitowy, a więc zawiera $2^8 = 256$ znaków. Znaki tego kodu ponumerowane są od 0 do $255 = 2^8 - 1$. Mamy tu znaki sterujące o numerach porządkowych w przedziałach 0..31 i 127..159, które nie należą do znaków graficznych. (W definicji typu Character (Barnes 1998) ich nazwy napisano kursywą). Znaki o numerach 0..127 odpowiadają znakom kodu ASCII (ISO 646), który w Adzie 83 jest standardowym typem znakowym. W Adzie 95 typ ten określono jako przestarzały i zastąpiono omawianym tu typem Character. Ponieważ typ ten jest z definicji typem wyliczeniowym, wszystkie operacje określone dla tego rodzaju typów mogą być stosowane do danych typu znakowego. W Adzie mamy pakiety Ada.Characters, Ada.Characters.Handling i Ada.Characters.Latin_1 zawierające podprogramy pozwalające na wykonywanie operacji na danych typu znakowego. Działanie tych funkcji bazuje na podziale zbioru znaków na podzbiory, które wymieniono w tabeli 3.5.

Wymienione w pierwszej kolumnie tabeli nazwy są identyfikatorami funkcji o wartościach typu Boolean i funkcje te służą do sprawdzenia, czy dany znak należy do odpowiedniego podzbioru. Użycie tych funkcji ilustruje poniższy przykład, w którym znaki sterujące i specjalne identyfikowane są jako stałe określone w pakiecie Ada.Characters.Latin_1.

```
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Characters.Handling; use Ada.Characters.Handling;
with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;

procedure Podzbiory_Latin_1 is
   Znak : Character;
begin
   Znak := Lc_A;
   if Is_Lower (Znak) then
      Put ("To jest mala litera ");
```

Tabela 3.5: Podzbiory zbioru znaków należących do typu Character

Określenie podzbioru	Numery porządkowe elementów podzbioru
Is_Control	0..31, 127..159
Is_Graphic	not Is_Control
Is_Lower	97..122, 223..246, 248..255
Is_Upper	65..90, 192..214, 216..222
Is_Letter	Is_Lower or Is_Upper
Is_Basic	65..90, 97..122, 198, 208, 222, 223, 230, 240, 254
Is_Digit	48..57
Is_Decimal_Digit	48..57
Is_Hexadecimal_Digit	48..57, 65..70, 97..102
Is_Alphanumeric	Is_Letter or Is_Digit
Is_Special	Is_Graphic and not Is_Alphanumeric
Is_ISO_648	0..127


```
11      Put (Znak);
12      New_Line;
13    end if;
14    Znak := 'A'; -- Wielkie litery nie mają nazw
15    if Is_Upper (Znak) then
16      Put ("To jest wielka litera ");
17      Put (Znak);
18      New_Line;
19    end if;
20    Znak := '1'; -- Cyfry nie mają nazw
21    if Is_Digit (Znak) then
22      Put ("To jest cyfra ");
23      Put (Znak);
24      New_Line;
25    end if;
26    Znak := Exclamation;
27    if Is_Special (Znak) then
28      Put ("To jest znak ");
29      Put (Znak);
30      New_Line;
31    end if;
32    Znak := Lf;
33    if Is_Control (Znak) then
34      Put ("Znak o numerze ");
35      Put (Integer'Image(Character'Pos(Znak)));
36      Put(" nie jest znakiem graficznym ");
37    end if;
38  end Podzbiory_Latin_1;
```

Wyniki działania tego programu są następujące:

```
To jest mala litera a
To jest wielka litera A
To jest cyfra 1
To jest znak !
Znak o numerze 10 nie jest znakiem graficznym
```

Ponadto, w języku Ada zdefiniowany jest typ Wide_Character reprezentujący wszystkie znaki ze wszystkich alfabetów za ziemi. Znaki te są zdefiniowane w kodzie o nazwie *UNICODE* (www.unicode.org 2000).

3.10 Deklaracje stałych i zmiennych

3

Ogólnie można powiedzieć, że dane dzielą się na *stałe* i *zmienne*. Stałymi są dane, które nie zmieniają swojej wartości w całym programie. Jeżeli programista chce używać stałej, która ma specjalne znaczenie dla rozwiązywanego zagadnienia, np. wybranej liczby, to w miejsce ciągu znaków reprezentujących tą stałą, powinien używać nazwy symbolicznej rozumianej jako pełny równoważnik tej stałej. Do przypisywania nazw (identyfikatorów) odpowiednim wartościom stałych służą deklaracje stałych, które składają się z identyfikatora stałej, po którym występuje dwukropek, następnie pojawia się słowo kluczowe constant, znak podstawienia „:=" i wartość stałej zakończona średnikiem. Poniżej podano kilka deklaracji stałych:

```
Pi: constant Float := 3.14159_26536;
Czestotliwosc: constant Float := 50.0; -- Hz
Pulsacja: constant Float := 2.0*Pi*Czestotliwosc; -- rad/s
Szerokosc_Ekranu_VGA: constant Integer := 640;
Wysokosc_Ekranu_VGA: constant Integer := 480;
Znak_Zero: constant Character := '0';
```

Zauważmy, że jeżeli w programie używana jest stała Czestotliwosc = 50.0, to w nowej, „amerykańskiej" wersji programu wymagającej Czestotliwosc = 60.0 należy dokonać tylko jednej zmiany polegającej na zmianie deklaracji stałej. Warto podkreślić, że taka zmiana oznacza zmianę wartości stałej, a nie roli tej stałej w programie, która powinna być wyrażona przez wybór znaczącego identyfikatora. Programista musi pamiętać, że jedna zmiana deklaracji stałej może spowodować zmianę innych wartości stałych, których deklaracje odwołują się do zmienionej stałej. W podanym przykładzie, zmiana wartości stałej Czestotliwosc powoduje zmianę stałej Pulsacja. Użycie nazwy Pi dla oznaczenia liczby 3.141_592_6536 zwiększa czytelność programu i zabezpiecza przed pomyłką w wypisywaniu długiego ciągu znaków tworzącego stałą Pi. Poza tym, jeżeli programista chce wielokrotnie stosować pulsację określaną zwykle jako $\omega = 2\pi f$, to lepiej zadeklarować stałą Pulsacja, tak jak zrobiono to w przykładzie, niż za każdym razem wykonywać dwa mnożenia więcej lub polegać na optymalizacji generowania kodu przez kompilator.

Reasumując deklaracje stałych

- ⇝ zwiększają czytelność programu,
- ⇝ zwiększają niezawodność programu,
- ⇝ skracają tekst programu,
- ⇝ ułatwiają wprowadzanie zmian w programie.

W żadnym wypadku nie należy sądzić, że stałe mogą należeć wyłącznie do typów prostych. Nic nie stoi na przeszkodzie, żeby stałe były stałymi dowolnie złożonego typu (rozdział 5).

W odróżnieniu od stałych, zmienne oznaczają dane, które mogą zmieniać swoje wartości w trakcie wykonywania programu. Podobnie jak stałe, wszystkie zmienne muszą być zdeklarowane, tzn. muszą otrzymać jednoznaczne nazwy – identyfikatory. Poza tym, każdej zmiennej przypisuje się zbiór, do którego muszą należeć wartości zmiennej, a co za tym idzie zbiór operacji, które mogą być wykonywane w tym zbiorze.

Deklaracje zmiennych mają najczęściej postać:

```
Identyfikator_Zmiennej : Indentyfikator_Typu;
```

Jeżeli chcemy stosować kilka zmiennych tego samego typu, to możemy stworzyć listę identyfikatorów tych zmiennych, w której poszczególne nazwy oddzielone są przecinkami i po dwukropku podać nazwę typu. Piszemy w związku z tym

```
Numer_Wiersza, Numer_Kolumny : Positive;
X, Y, Z : Float;
Znak_Sterujacy, Litera, Cyfra : Character;
D_Pracy, D_Swieto : Dzien_Tygodnia;
Poczatek, Koniec : Boolean;
```

Możemy jednak w deklaracjach zmiennych umieścić więcej informacji, które mogą zwiększyć niezawodność i czytelność naszego programu. Mianowicie, w deklaracji zmiennej możemy podać zakres wartości jakie deklarowana zmienna ma przyjmować, a także możemy nadać jej wartość początkową.

Ilustrują to następujące deklaracje:

```
Calkowita : Integer range −100 .. 100 := 1;
Rzeczywista : Float range −1.0 .. 1.0 := 0.0;
Wielka_Litera: Character range 'A' .. 'Z' := 'B';
```

Zwróćmy uwagę, że zmienne Calkowita, Rzeczywista i Wielka_Litera są odpowiednio typów Integer, Float i Character, natomiast po słowie kluczowym range podano zakresy wartości jakie te zmienne mogą przyjmować. Ponadto, na końcu każdej deklaracji dokonano podstawienia wartości początkowej. Należy podkreślić, że podstawienia te wykonywane są podczas kompilacji programu, a nie podczas jego wykonywania.

Jak pamiętamy, ograniczenie zakresu zmiennych można również osiągnąć stosując deklaracje typów i podtypów. Moglibyśmy napisać tak:

```
subtype Zakres is Integer range −100 .. 100;
subtype Odcinek is Float range −1.0 .. 1.0;
subtype Od_A_Do_Z is Character range 'A' .. 'Z';
Calkowita : Zakres := 1;
Rzeczywista : Odcinek := 0.0;
Wielka_Litera : Od_A_Do_Z := 'B';
```

W porównaniu z poprzednimi deklaracjami, mamy pewną różnicę polegającą na tym, że określiliśmy odpowiednie podtypy, co pozwala na odwoływanie się do

nazw tych podtypów w innych deklaracjach zmiennych np. parametrów formalnych podprogramów, albo zmiennych lokalnych podprogramów, lub zmiennych definiowanych wewnątrz instrukcji declare.

Jeżeli napisalibyśmy, zapewne przez pomyłkę,

```
subtype Zakres is Integer range 101 .. 100;
```


to zbiór wartości podtypu Zakres jest zbiorem pustym i podczas kompilacji otrzymamy ostrzeżenie informujące, że przy wykonywaniu programu wystąpi błąd przekroczenia ograniczeń (CONSTRAINT_ERROR). To samo ostrzeżenie zostanie wypisane, jeżeli wartość początkowa zmiennej jest spoza deklarowanego zakresu. Kompilator Ady nie musi nadawać domyślnych wartości deklarowanym zmiennym[6]. W celu świadomego ich inicjowania mamy do dyspozycji mechanizm nadawania wartości początkowych przy deklaracji zmiennych. Korzystanie z tego mechanizmu pozwala na unikanie trudnych do wykrycia błędów powstających przez użycie niezainicjowanych zmiennych w wyrażeniach, ponieważ w takiej sytuacji musimy polegać na tym jak został zaprojektowany nasz kompilator. Zjawisko to ilustruje program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Deklaracje_Zmiennych is
6     Calkowita : Integer range -100 .. 100;
7     Rzeczywista : Float range -1.0 .. 1.0 := 0.0;
8     Wielka_Litera : Character range 'A' .. 'Z' := 'B';
9  begin
10    Put (Calkowita);
11    New_Line;
12    Put (Rzeczywista);
13    New_Line;
14    Put (Wielka_Litera);
15 end Deklaracje_Zmiennych;
```

Wykonanie tego programu dało wyniki:

```
4198900
0.00000E+00
B
```

Ten sam program skompilowany przy pomocy kompilatora Ady'95 firmy Aonix znajdującego się na płycie dołączonej do książki Barnes'a (Barnes 1998) dał wyniki:

```
0
0.00000E+00
B
```

Zwróćmy uwagę na to, że zmienna Calkowita została zainicjowana inaczej przez każdy z kompilatorów. Dlatego, jeżeli jest to możliwe zalecamy świadome inicjowanie zmiennych przy ich deklaracji.

[6] Zmienne typu wskaznikowego (rozdział 8) są **zawsze** zainicjowane w taki sposób, że nie wskazują na żaden obiekt (wartością null).

3.11 Wyrażenia

Wyrażenia tworzymy ze skończonej ilości argumentów i operatorów. Argumentami wyrażeń mogą być następujące obiekty:

- identyfikatory,
- literały,
- konwersje typów,
- wyrażenia kwalifikowane,
- wywołania funkcji.

Identyfikatorami są nazwy zmiennych, lub stałych takie jak Calkowita, Rzeczywista, Wielka_Litera, Pi, Pulsacja, Znak_Zero, Exclamation itp.

Przykładami literałów są następujące ciągi znaków: 3.14159_26356, 50.0, 640, 16#1FA1#, 'A', '0', Czerwony, "Ada_95", przy czym pierwszych pięć należy do literałów liczbowych, dwa następne są literałami znakowymi, następny jest literałem wyliczeniowym, a ostatni literałem napisowym.

Konwersje typów wymagają nieco szerszego omówienia. Jak wspomniano w Adzie obowiązuje tzw. silna typizacja i w wyrażeniach nie wolno używać argumentów różnych typów. Dla przykładu przypuśćmy, że chcemy obliczyć rozwinięcie na szereg Fouriera pewnej funkcji okresowej $f(\cdot)$ o okresie $T = 2\pi/\omega$, gdzie ω oznacza pulsację podstawową. Przy obliczaniu N-tej sumy częściowej tego szeregu (Papoulis 1988, Schwarz i Shaw 1975):

$$f_N(\cdot) = a_0 + \sum_{k=1}^{N} (a_k \cos k\omega t + b_k \sin k\omega t) \tag{3.1}$$

gdzie k oznacza numery harmonicznych funkcji $f(\cdot)$, a_0, a_k, b_k, nazywane są współczynnikami Fouriera, należy m.in. wyznaczać wartości wyrażeń $\cos k\omega t$ i $\sin k\omega t$.

Jeżeli zadeklarujemy zmienne:

```
Numer_Harm : Positive;
Podstawowa : Float;
Czas : Float;
Kat : Float;
```

i przyjmiemy, że do naszego programu dołączamy (porównaj pierwszy program w Adzie, strona 22) pakiet biblioteczny Ada.Numerics.Elementary_Functions zawierający potrzebne funkcje trygonometryczne, to pisząc

```
Kat := Numer_Harm*Podstawowa*Czas;
```

popełniamy błąd, ponieważ zmienne Numer_Harm i Podstawowa są różnych typów. Aby uzyskać poprawne wyrażenie musimy dokonać jawnej konwersji typu zmiennej Numer_Harm na typ Float, co zapisujemy

```
Kat := Float(Numer_Harm)*Podstawowa*Czas;
```

Podobnie nie wolno pisać abs (−1), albo Sqrt(2) i należy te wywołania funkcji napisać w postaci odpowiednio abs (−1.0) i Sqrt(2.0).

Niektóre własności konwersji typów ilustruje wykonanie następującego programu:


```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Konwersja is
6     Calkowita : Integer := -1;
7     Rzeczywista : Float := -0.45;
8  begin
9     Put ("Konwersja z Integer na Float");
10    New_Line;
11    Put (Calkowita, 2);
12    Put (" na ");
13    Put (Float(Calkowita), 2, 5, 0);
14    New_Line;
15    Put ("Konwersja z Float na Integer ");
16    New_Line;
17    Put (Rzeczywista, 2, 5, 0);
18    Put (" na ");
19    Put (Integer(Rzeczywista), 2);
20    New_Line;
21    Rzeczywista := -0.5;
22    Put (Rzeczywista, 2, 5, 0);
23    Put (" na ");
24    Put (Integer(Rzeczywista), 2);
25    New_Line;
26    Rzeczywista := -0.55;
27    Put (Rzeczywista, 2, 5, 0);
28    Put (" na ");
29    Put (Integer(Rzeczywista), 2);
30 end Konwersja;
```

który wypisał poniższe dane:

```
Konwersja z Integer na Float
-1 na -1.00000
Konwersja z Float na Integer
-0.45000 na 0
-0.50000 na -1
-0.55000 na -1
```

Zauważmy, że przy konwersji z typu zmiennopozycyjnego na typ całkowity występuje zaokrąglenie liczby rzeczywistej do najbliższej wartości całkowitej.

Uwaga!

Częste stosowanie konwersji jawnej może wskazywać na to, że przyjęto nieodpowiednie typy danych!

Konwersja typu różni się od kwalifikacji typu. W pierwszym przypadku zmieniany jest typ, natomiast w drugim przypadku następuje tylko sprawdzenie, czy zmienna kwalifikuje się do innego typu. Jeżeli mamy deklaracje:

```
F : Float;
I : Integer;
```

to konwersję F na typ Positive zapisujemy

```
Positive (F),
```


podczas gdy kwalifikację I do typu Positive zapisujemy w postaci:

```
Positive'(I).
```

Przy podanej konwersji następuje zmiana typu zmiennej F na typ Integer (Positive jest podtypem określonym na bazie Integer) i sprawdzane jest czy po konwersji wartość jest dodatnia, natomiast drugi zapis powoduje wyłącznie sprawdzenie, czy I jest dodatnie. Dla odróżnienia kwalifikacja oznaczana jest apostrofem, który wstawiamy po nazwie typu, do którego kwalifikujemy zmienną.

Wyrażenia kwalifikowane mają postać:

```
Dzien_Roboczy'(Poniedziałek)
Kolor'(Niebieski),
```

a przykładami atrybutów są:

```
Boolean'Last
Integer'First.
```

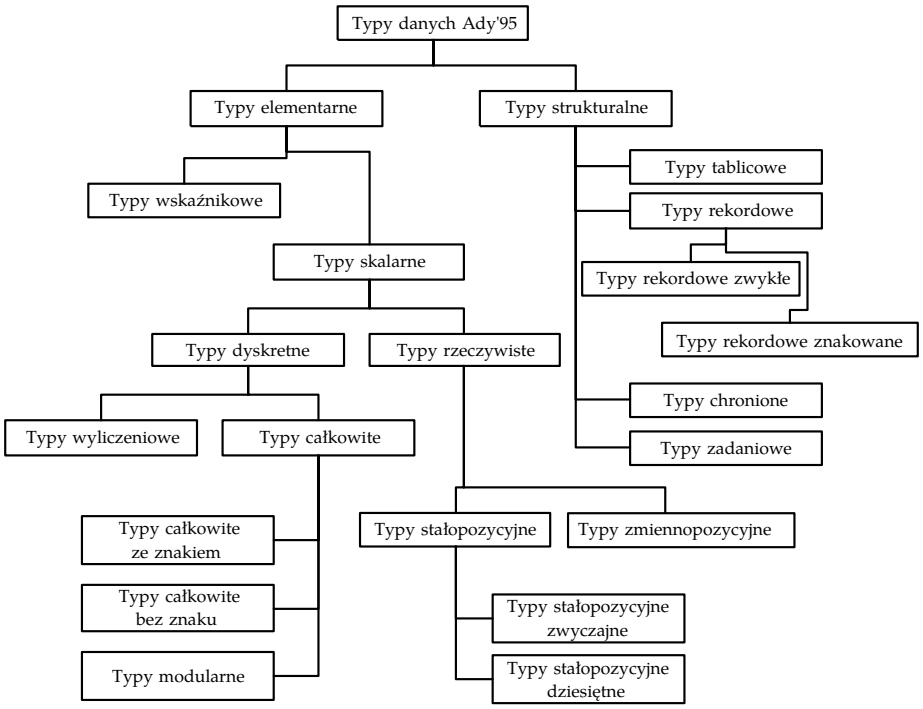
Ostatnim wymienionym rodzajem argumentów wyrażeń są wywołania funkcji. Dokładniejsze omówienie podprogramów, czyli funkcji i procedur można znaleźć w rozdziale 6, natomiast w tym miejscu wystarczy wiedzieć, że przykładami wywołań funkcji są: Sin(Kat), Cos(Kat), abs(1) i Sqrt(2.0). Należy pamiętać, że wartość obliczana przez funkcję musi być typu zgodnego z innymi argumentami wyrażenia.

Na zakończenie tego podpunktu warto dodać, że mamy zdefiniowane wstępnie dwa atrybuty Max i Min działające na dwa argumenty dowolnego typu skalarnego, lub podtypu. Jak sugerują ich nazwy, atrybuty te służą odpowiednio do obliczania większego, albo mniejszego z argumentów. Mamy więc:

```
Integer'Max (0, 1) = 1
Integer'Min (0, −1) = −1
Float'Max (abs(−1.0), abs(1.0)) = 1.00000E+00
Float'Min (Sqrt(2.0), 2.0)) = 1.41421E+00
```

3.12 Klasyfikacja typów

Na zakończenie tego rozdziału podajemy klasyfikację typów danych Ady (Barnes 1998, Smith 1996). Wszystkie typy dzielimy na typy elementarne i strukturalne. Typy elementarne dzielą się na typy skalarne, albo proste i typy wskaźnikowe. W grupie typów skalarnych wyróżniamy typy dyskretne i rzeczywiste. Te pierwsze



Rysunek 3.1: Klasyfikacja typów

dzielą się na typy wyliczeniowe i typy całkowite, które z kolei dzielą się na typy całkowite ze znakiem i typy całkowite bez znaku. Wśród typów rzeczywistych mamy typy zmiennopozycyjne i stałopozycyjne, które dzielą się jeszcze na typy stałopozycyjne zwyczajne i dziesiętne.

Typy strukturalne dzielimy na cztery grupy: tablicowe, rekordowe, chronione i zadaniowe.

Schemat tej klasyfikacji typów Ady'95 pokazano na rysunku 3.1.

Rozdział 4

Instrukcje

Jak pamiętamy z rozdziału 1, formalny opis akcji wykonywanej przez komputer nazywamy *instrukcją*. Efektem wykonania instrukcji jest przekształcenie stanu obliczeń, przy czym *stanem obliczeń* nazywamy wartości wszystkich zmiennych programu. Jeżeli kilka instrukcji ma być wykonanych jedna po drugiej, to mamy do czynienia z *ciągiem instrukcji*. Ciąg składający się z N instrukcji zapisujemy w postaci:

```
Instrukcja_1;
Instrukcja_2;
...
...
Instrukcja_N;
```

Zauważmy, że każda instrukcja kończona jest średnikiem.

Poza tym, każda instrukcja może być poprzedzona *etykietą*, która jest identyfikatorem ujętym w podwójne nawiasy kątowe. Można więc pisać:

```
<< Etykieta >> Instrukcja;
```

Poprzedzenie instrukcji etykietą umożliwia wykonanie skoku do tej instrukcji. Skok ten realizowany jest przez wykonanie odpowiedniej instrukcji goto. W naszym przypadku skok taki ma postać:

```
goto Etykieta;
```

Instrukcję skoku pozostawiono w Adzie po to, aby ułatwić tłumaczenie programów napisanych z użyciem tej instrukcji i wyrażonych w językach takich jak Fortran, czy Pascal na programy Ady.

Przepływ sterowania

Kolejność wykonywania instrukcji w programie nazywana jest *przepływem sterowania* (ang. *flow control*). W danej chwili komputer steruje (kontroluje) wykonaniem jednej instrukcji. Mówi się, że po wykonaniu pewnej instrukcji sterowanie przekazywane jest do następnej.

Najprostszą sytuację mamy w przypadku *sekwencyjnego przepływu* przekazywania *sterowania*. W przypadku gdy wykonywana jest instrukcja skoku, to sterowanie zostaje przekazane do instrukcji opatrzonej etykietą podaną w instrukcji skoku. Nie można jej stosować do przekazywania sterowania do wnętrza instrukcji warunkowych, i pętli lub pomiędzy rozgałęzieniami instrukcji warunkowych. Można jej natomiast użyć do wyjścia z pętli lub z bloku.

Jeżeli jednak Czytelnik lub Czytelniczka pisząc pewien program uzna, że zastosowanie tej instrukcji jest niezbędne, to można przyjąć ze 100% pewnością, że ten program jest źle przemyślany. Mimo tego, że napisaliśmy już wiele tysięcy linii kodu w języku Ada nigdy nie zostaliśmy zmuszeni do zastosowania tej instrukcji, ba, nawet nie poczuliśmy takiej potrzeby!

4

Instrukcje można podzielić na *instrukcje proste* i *instrukcje złożone*. Instrukcje złożone składają się z wewnętrznych ciągów instrukcji, których wyrazy mogą być instrukcjami prostymi, albo złożonymi. W związku z tym mówimy o zagnieżdżaniu instrukcji. W Adzie nie ma ograniczenia ilości poziomów zagnieżdżenia, ale jest oczywiste, że zbyt głębokie zagnieżdżanie zmniejsza czytelność programów.

Instrukcje proste są instrukcjami sekwencyjnymi, czyli wykonywanymi w kolejności ich zapisu, albo rozgałęzieniami. Zazwyczaj rozgałęzienia występują na końcu ciągu instrukcji sekwencyjnych. W zależności od rodzaju instrukcji rozgałęzienia wykonywana jest następna instrukcja, która może być zapisana w zupełnie innym miejscu programu.

Instrukcje złożone mogą być jednocześnie instrukcjami prostymi i rozgałęzieniami. Instrukcje złożone ujmuje się w nawiasy syntaktyczne, którymi są odpowiednie słowa kluczowe Ady, przy czy ciągi instrukcji nie są ujmowane w takie nawiasy. Do instrukcji złożonych należą instrukcje warunkowe, instrukcje pętli oraz instrukcja declare, które omawiane są w dalszych punktach tego rozdziału.

4.1 Instrukcja pusta

Instrukcja pusta składa się z jednego słowa zastrzeżonego

```
null;
```

i oznacza „nie rób nic”.

Czytelnik i Czytelniczka zapewne dziwią się po co w ogóle pisać taką instrukcję? Odpowiedź na to pytanie należy poprzedzić następującą dygresją: W różnych dokumentacjach, w szczególności w dokumentacjach angielskich nie ma stron pustych. Jeżeli z jakiegoś względu na danej stronie nie ma żadnego tekstu, to występuje na niej napis *this page is intensionally blank*[1]. Celem takiego postępowanie jest niepozostawianie czytelnikom wątpliwości, że dokumentacja jest niekompletna, że coś zostało zapomniane.

Dokładnie taki sam cel przyświecał twórcom języka Ada. Jeżeli chcesz Czytelniku, by dana procedura nic nie robiła, to napisz to! Wtedy czytający Twój program (a nawet Ty sam po pewnym czasie) będzie pewien, że tak miało być, i nie jest to wynikiem zapomnienia.

[1] Ta strona jest celowo pozostawiona pusta.

4.2 Instrukcja podstawienia.

Najbardziej elementarną instrukcją jest *instrukcja podstawienia* (nazywaną też instrukcją przypisania), która ma następującą postać:

Desygnator := Wyrażenie;

Instrukcja podstawienia powoduje wykonanie następujących akcji:

- ⇝ wyznaczenie wartości desygnatora opisującego zmienną,
- ⇝ wyznaczenie wartości wyrażenia,
- ⇝ zastąpienie wartości zmiennej opisanej desygnatorem przez wartość wyrażenia.

Poniżej podano przykład kilku instrukcji podstawienia:

```
Numer_Harm := 1;
Podstawowa := 2.0*Pi;
Czas := 0.0;
Kat := Float(Numer_Harm)*Podstawowa*Czas;
Wyrazy(Numer_Harm) := A(Numer_Harm)*Sin(Kat) +
                      B(Numer_Harm)*Cos(Kat);
```

Desygnatorami są tu identyfikatory zmiennych Numer_Harm, Podstawowa, Czas, Kat natomiast desygnator Wyrazy(Numer_Harm), oznacza składową tablicy jednowymiarowej Wyrazy. Tablice należą do typów strukturalnych, które omawiane są w następnym rozdziale.

Należy pamiętać, że poprzednie wartości zmiennych opisywanych przez odpowiednie desygnatory, są tracone. Z opisu akcji składających się na wykonanie instrukcji podstawienia wynika, że każda zmienna występująca w tej instrukcji musi mieć wcześniej nadaną wartość. Kolejność wykonywania kilku instrukcji podstawienia może mieć istotne znaczenie. Dla przykładu rozważmy dwa ciągi instrukcji:

Pierwszy ciąg:

```
i := 0;
i := i + 1;
j := 2 * i;
```

Drugi ciąg:

```
i := 0;
j := 2 * i;
i := i + 1;
```

Po wykonaniu pierwszego ciągu zmienne i, j przyjmują wartości:

i = 1, j = 2, a po wykonaniu drugiego ciągu – i = 1, j = 0.

Z tego wynika, że wymiana wartości zmiennych i, j nie może być wykonana przez następującą parę instrukcji:

```
i := j;
j := i;
```

Aby poprawnie wykonać to zadanie należy użyć zmiennej pomocniczej k dla przechowania wartości jednej ze zmiennych i wykonać następujący ciąg instrukcji:

```
k := i;
i := j;
j := k;
```

4

4.3 Instrukcja declare

W Adzie, podobnie jak w innych współczesnych językach programowania wysokiego poziomu, istnieje ścisłe rozróżnienie deklaracji i instrukcji. Deklaracje określają nowe identyfikatory, a instrukcje zmieniają stan obliczeń. Jest zgodne z intuicją, że najpierw deklarujemy pewien obiekt, a następnie przy pomocy instrukcji wykonujemy z jego udziałem pewne operacje. W związku z tym deklaracje i instrukcje występują w oddzielnych częściach programu. Czytelnik zauważył zapewne, że w naszym pierwszym programie po przypisaniu mu nazwy Pierwiastki_Trojmianu zadeklarowano zmienne, a następnie po słowie begin podano instrukcje określające jak mają być wykonywane obliczenia. W Adzie można w dowolnym miejscu programu deklarować nowe zmienne i podawać pewien ciąg instrukcji. Do tego celu służy pojęcie *bloku*. Blok zaczynamy słowem kluczowym declare, po którym podajemy potrzebne deklaracje, następnie piszemy słowo begin, piszemy instrukcje wykonywane wewnątrz bloku i kończymy go słowem end. Blok zapisujemy w postaci:

```
declare
   Deklaracje;
begin
   Instrukcje;
end;
```

Blok jest instrukcją i może zawierać inne instrukcje declare, a te z kolei mogą zawierać następne bloki itd. bez ograniczenia głębokości zagnieżdżeń[2]. Jeżeli wykonywana jest instrukcja declare, to kreowane są obiekty deklarowane w bloku i wykonywane są instrukcje bloku. Należy zwrócić uwagę na to, że po zakończeniu bloku wszystkie obiekty w nim zadeklarowane znikają i nie można się do nich odwołać. Weźmy pod uwagę następujący, krótki program:

```
1 with Ada.Text_IO; use Ada.Text_IO; with
2 Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4 procedure Blok_Zasieg is
5     A : Positive := 1;
6     B : Positive := A;
7 begin
```

[2]Co nie znaczy, że należy stosować instrukcje wielokrotnie zagnieżdżane. Trzeba pamiętać, że zbyt duża liczba zagnieżdżeń ogranicza czytelność programów. W przypadku, w którym należałoby zastosować więcej niż kilka poziomów zagnieżdżeń należy stosować procedury (rozdział 6)

```
8     Put ("Wartości zmiennych przed wejściem do bloku");
9     New_Line;
10    Put ("A = ");
11    Put (A, 2);
12    Put (" B = ");
13    Put (B, 2);
14    declare
15       A : Integer := -1;
16       B : Integer := -1;
17    begin
18       New_Line;
19       Put ("Wartości zmiennych wewnątrz bloku");
20       New_Line;
21       Put ("A = ");
22       Put (A, 2);
23       Put (" B = ");
24       Put (B, 2);
25    end;
26    New_Line;}
27    Put ("Wartości zmiennych po wyjściu z bloku");
28    New_Line;
29    Put ("A = ");
30    Put (A, 2);
31    Put (" B = ");
32    Put (B, 2);
33 end Blok_Zasieg;
```

Wykonanie tego programu dało wyniki:

```
Wartości zmiennych przed wejściem do bloku
A = 1 B = 1
Wartości zmiennych wewnątrz bloku
A = -1 B = -1
Wartości zmiennych po wyjściu z bloku
A = 1 B = 1
```

Zauważmy, że w podanym programie te same identyfikatory A i B oznaczają zmienne globalne programu i zmienne bloku. Zmienne te nie są tymi samymi obiektami. Nazwy A, B zadeklarowane w bloku *przesłoniły* zmienne zadeklarowane w programie, co zilustrowano wypisując odpowiednie wartości. Poza tym należy tu zwrócić uwagę na to, że przy nadawaniu wartości początkowych zastosowano tzw. zasadę liniowej interpretacji deklaracji, z której wynika, że nie musimy pisać B : Positive := 1, ponieważ kompilator wcześniej nadał zmiennej A wartość 1 i mógł przypisać zmiennej B wartość A.

Warto wspomnieć, że nadawanie wartości zmiennym w każdym miejscu programu, w tym także w instrukcji declare może polegać na obliczeniu wyrażenia dowolnej złożoności, w tym wywołanie podprogramu.

Z pojęciem bloku wiążą się pojęcia *zasięgu* i *widzialności*. Zasięgiem nazywamy fragment tekstu programu, w którym obiekt ma znaczenie. W przypadku bloku, zasięg obiektów (zmiennych, lub stałych) deklarowanych w bloku obejmuje obszar od ich deklaracji do końca bloku. Obiekt jest widzialny w swoim zasięgu tzn. można się do niego odwoływać, chyba że zostanie przesłonięty

przez nową deklarację obiektu o tej samej nazwie. Mamy następujące reguły widzialności:

- ⇝ obiekt nie jest widzialny w swojej własnej deklaracji,
- ⇝ obiekt jest zasłaniany przez deklarację obiektu o tym samym identyfikatorze poczynając od początku tej deklaracji.

4.4 Instrukcje warunkowe

4

Bardzo często algorytmy dające przepis na wykonanie pewnego zadania zawierają części (podzadania) wykonywane w zależności od okoliczności. Wyboru odpowiedniego podzadania dokonać można na podstawie badania wartości wyrażenia logicznego, albo na podstawie sprawdzenia wartości pewnej zmiennej. Do tego celu mamy w Adzie instrukcje warunkowe if i case, przy czym w przypadku pierwszej, wybór zależy od wartości wyrażenia logicznego, czyli warunku, a w przypadku drugiej realizowany wariant algorytmu określa wartość danych. Najpierw zajmiemy się instrukcją warunkową if.

4.5 Instrukcja `if`

Opis instrukcji warunkowej w notacji EBNF wygląda następująco:

```
if_then_else_statement ::=
    if condition then
        sequence_of_statements
    {
    elsif condition then
        sequence_of_statements
    }
    [
    else
        sequence_of_statements
    ]

sequence_of_statements ::= statement {statement}
condition ::= Boolean_expression;
```

Najprostszą formę instrukcji if zapisujemy w postaci:

```
if Warunek then
   Ciag_Instrukcji;
end if;
```

W podanym tekście Warunek jest wyrażeniem logicznym (typu Boolean) i jeżeli ma ono wartość True, to wykonywany jest Ciag_Instrukcji[3], a w przeciwnym przypadku wykonywana jest instrukcja występująca po nawiasie zamykającym instrukcję if, czyli po end if.

[3] Zwróćmy uwagę, że Ciag_Instrukcji składa się z co najmniej jednej instrukcji, przy czym może to być instrukcja null (rozdział 4.1)

Może się zdarzyć, że realizowany algorytm wymaga podjęcia akcji, alternatywnej w stosunku do akcji podejmowanej, gdy Warunek jest prawdziwy. W takiej sytuacji możemy zastosować następującą postać instrukcji if:

```
if Warunek then
    Ciag_Instrukcji;
else
    Inny_Ciag_Instrukcji;
end if;
```

W tym przypadku Inny_Ciag_Instrukcji jest wykonywany, gdy Warunek przyjmuje wartość False. Zwróćmy uwagę, że tutaj akcja opisana przez Ciag_Instrukcji nie jest podejmowana. Mamy więc do czynienia z rozgałęzieniem, które jest realizowane wewnątrz instrukcji if. Jaka instrukcja zostanie wykonana po wyjściu z tej instrukcji zależy od instrukcji podanych w odpowiednich ciągach. W szczególności może tam być instrukcja skoku i wtedy nastąpi przeniesienie akcji do miejsca oznaczonego przez związaną z tym skokiem etykietę. Należy jednak pamiętać, że używanie instrukcji skoku świadczy o złym stylu programowania[4]. Zobacz też (Green 2001).

Jeżeli w ciągu Inny_Ciag_Instrukcji mamy jako pierwszą instrukcję if, to zamiast pisać niezbyt ładnie:

```
if Warunek then
    Ciag_Instrukcji;
else
    if Warunek_1 then
        Ciag_Instrukcji_1;
    else
        Ciag_Instrukcji_2;
    end if;
end if;
```

możemy użyć konstrukcji ze słowem kluczowym elsif, a więc

```
if Warunek then
    Ciag_Instrukcji;
elsif Warunek_1 then
    Ciag_Instrukcji_1;
else
    Ciag_Instrukcji_2;
end if;
```

Doszliśmy w ten sposób do ogólnej formy instrukcji if, którą zapisujemy w postaci:

```
if Warunek_1 then
    Ciag_1;
elsif Warunek_2 then
    Ciag_2;
elsif Warunek_3 then
    Ciag_3;
```

[4] W znanym i cenionym w niektórych środowiskach języku Modula-2 (Arendt, Postół i Zajączkowski 1988), podobnie jak i w języku Java (Eckel 2001) nie ma instrukcji goto.

```
...
...

else
   Ciag_Ostatni;
end if;
```

Jeżeli wyrażenie Warunek_1 = True, to wykonywany jest Ciag_1, następne warunki nie są sprawdzane i instrukcja if jest kończona. Jeżeli Warunek_1 = False, to sprawdzane są następne warunki do momentu napotkania warunku o wartości True. Wtedy wykonywany jest odpowiedni ciąg instrukcji i instrukcja if jest kończona. Jeżeli żaden z warunków nie jest spełniony, to wykonywany jest ciąg instrukcji występujący po słowie else. Ogólną postać instrukcji if zastosowano w następującej funkcji *Signum* (funkcje i procedury omawiane są w rozdziale dotyczącym podprogramów):

```
1  function Signum (X : Float ) return Float is
2  begin
3      if X < 0.0 then
4          return -1.0;
5      elsif X > 0.0 then
6          return 1.0;
7      else
8          return 0.0;
9      end if;
10 end Signum;
```

Pamiętaj!

Pisanie wyrażenia w instrukcji warunkowej, przy założeniu, że a jest wyrażeniem logicznym lub po prostu zmienną typu Boolean w postaci:

```
if a = True then
   ...
end if;
```

świadczy o fatalnym stylu programowania, chociaż jest prawidłowe. Bo dlaczego nie pisać a /= False?

Powyższa instrukcja powinna być napisana następująco:

```
if a then
   ...
end if;
```

Uzupełnienie – Wejście-wyjście w przypadku typów wyliczeniowych

Tutaj odwołujemy się do konkretyzacji pakietów ogólnych omówionych w rozdziale 13. Konkretyzacja będzie dotyczyć pakietu Ada.Text_IO.Enumeration_IO, który jest zdefiniowany następująco:

```
1  private generic
2      type Enum is (<>);
3
4  package Ada.Text_IO.Enumeration_IO is
5
```

```
6     Default_Width : Field := 0;
7     Default_Setting : Type_Set := Upper_Case;
8
9     procedure Get (File : in File_Type; Item : out Enum);
10    procedure Get (Item : out Enum);
11
12    procedure Put
13      (File : in File_Type;
14       Item : in Enum;
15       Width : in Field := Default_Width;
16       Set : in Type_Set := Default_Setting);
17
18    procedure Put
19      (Item : in Enum;
20       Width : in Field := Default_Width;
21       Set : in Type_Set := Default_Setting);
22
23    procedure Get
24      (From : in String;
25       Item : out Enum;
26       Last : out Positive);
27
28    procedure Put
29      (To : out String;
30       Item : in Enum;
31       Set : in Type_Set := Default_Setting);
32
33 end Ada.Text_IO.Enumeration_IO;
```

Jego konkretyzacja dla typu

```
type Nazwy_Miesiecy is (Styczen, Luty, Marzec,
                        Kwiecien, Maj, Czerwiec,
                        Lipec, Sierpien, Wrzesien,
                        Pazdziernik, Listopad, Grudzien);
```

będzie wyglądała następująco

```
package Miesiac_IO is new Ada.Text_IO.Enumeration_IO
        (Enum => Miesiac)
```

a dla typu standardowego Boolean

```
package Boolean_IO is new Ada.Text_IO.Enumeration_IO (Boolean)
```

zaś ewentualne użycie:

```
Miesiac_IO.Put (Item => Miesiac'First, 20, Lower_Case);
Boolean_IO.Put (Index > 23);
```

4.6 Instrukcja case

Instrukcja case może być traktowana jako uogólnienie instrukcji if. Instrukcja ta jest też nazywana instrukcją wyboru, ponieważ stosowana jest wtedy, gdy

chcemy wykonać jeden z możliwych do wyboru ciągów instrukcji. Wybór konkretnego ciągu do wykonania zależy od wartości *wyrażenia wyboru*, nazywanego też *selektorem*, przy czym wartości selektora muszą być typu dyskretnego. Ogólna postać tej instrukcji jest następująca:

```
case Wyrazenie_Wyboru is
   when Lista_Wyboru_1 => Ciag_Instrukcji_1;
   when Lista_Wyboru_2 => Ciag_Instrukcji_2;
   ...
   ...
   when Lista_Wyboru_Ostatnia => Ciag_Instrukcji_Ostatni;
   when others => Ciag_Instrukcji_Inny; -- nie musi występować
end case;
```


Elementy umieszczone w listach wyboru muszą być tego samego typu jak typ wyrażenia wyboru, przy czym postać listy dostosowuje się do rozwiązywanego problemu.

Można tę listę podać w formie skończonego ciągu wyrażeń statycznych, w którym kolejne wyrazy oddziela się znakiem | np. Czerwony | Niebieski | Zielony[5].

Lista może być też zakresem wartości typu dyskretnego, przy czym można stosować następujące warianty: Wyrazenie_Lewe .. Wyrazenie_Prawe; np. Czerwony .. Zielony.

Odwołanie do atrybutu zakresu np. Kolor'Range.

Nazwa podtypu dyskretnego z opcjonalnym zawężeniem np. Natural range 1..100.

Reasumując, należy przestrzegać poniższych reguł:

1. Zbiory określone przez listy wyboru muszą być wzajemnie rozłączne.
2. Suma tych zbiorów musi być całym zbiorem wartości związanym z typem selektora.

Jeżeli występuje wariant oznaczony słowem kluczowym others, to musi być umieszczony na końcu.

Wszystkie wyrażenia określające listy wyboru muszą być statyczne, co oznacza, że podzbiory określone tymi listami muszą być określone podczas kompilacji.

Sposób użycia instrukcji wyboru ilustrują dwa podane niżej programy.

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Musztra is
4      type Zwroty is (Bacznosc, W_Lewo, W_Prawo, W_Tyl);
5      subtype Okrzyk is String (1..14);
6      Rozkaz : Zwroty := Bacznosc;
7
8      function Podaj_Rozkaz (X : Zwroty) return Okrzyk is
9          Komenda : Okrzyk;
10     begin
11         New_Line;
12         case X is
```

[5] W niektórych implementacjach zamiast znaku | można używać znaku wykrzyknika (!)

```
13              when W_Lewo =>
14                  Komenda := "W lewo zwrot ";
15              when W_Prawo =>
16                  Komenda := "W prawo zwrot ";
17              when W_Tyl =>
18                  Komenda := "W tył zwrot ";
19              when Bacznosc =>
20                  Komenda := "Baczność ";
21          end case;
22          return Komenda;
23      end Podaj_Rozkaz;
24
25  begin
26      Put (Podaj_Rozkaz (Rozkaz));
27      Rozkaz := W_Lewo;
28      Put (Podaj_Rozkaz (Rozkaz));
29      Rozkaz := W_Prawo;
30      Put (Podaj_Rozkaz (Rozkaz));
31      Rozkaz := W_Tyl;
32      Put (Podaj_Rozkaz (Rozkaz));
33  end Musztra;
```

```
1   with Ada.Text_Io; use Ada.Text_Io;
2
3   procedure Czytanie_Znakow is
4       Znak : Character;
5   begin
6       Put ("Wprowadź znak z klawiatury ");
7       New_Line;
8       loop
9           Get (Znak);
10          case Znak is
11              when '0' | '1' | '2' | '3' | '4' | '5' |'6' | '7' | '8' | '9' =>
12                  Put ("To jest cyfra dziesiętna = ");
13                  Put (Znak);
14                  New_Line;
15              when 'A' .. 'Z' =>
16                  Put ("To jest wielka litera = ");
17                  Put (Znak);
18                  New_Line;
19              when 'a' .. 'z' =>
20                  Put ("To jest mala litera = ");
21                  Put (Znak);
22                  New_Line;
23              when '_' =>}
24                  Put ("To jest podkreślnik = ");
25                  Put (Znak);
26                  New_Line;
27              when others =>
28                  Put ("To jest inny znak ");
29          end case;
30          exit when not Znak in '0' .. '9' or
31                        (Znak in 'A' .. 'Z') or
32                        (Znak in 'a' .. 'z') or
```

```
33                  Znak = '_';
34     end loop;
35 end Czytanie_Znakow;
```

Instrukcja case powoduje wykonanie następujących akcji:

- ⇝ Wartościowanie selektora.
- ⇝ Sprawdzenie do którego z podzbiorów wartości określonych listami wyboru należy wartość selektora.
- ⇝ Wykonanie ciągu instrukcji odpowiadającego wybranej w punkcie 2 liście.

Jeżeli w punkcie 2 nie wybrano żadnego wariantu, to zgłaszany jest wyjątek CONSTRAINT_ERROR, chociaż większość współczesnych kompilatorów zgłasza zastrzeżenia w przypadku, gdy nie wszystkie etykiety wyboru są wykorzystane, i w związku z tym taki błąd nie powinien się zdarzyć. Jeżeli dla danego selektora program nie powinien wykonać żadnej szczególnej operacji, to powinien wykonać instrukcję pustą (rozdział 4.1).

Uzupełnienie

Skalarnym typem danych nazywamy typ, w którym wartości są uporządkowane i każda jest niepodzielna.

Dyskretnym typem danych nazywamy typ skalarny, w którym każda wartość (z wyjątkiem pierwszej) ma jednoznacznie określonego poprzednika i każda wartość (z wyjątkiem ostatniej) ma jednoznacznie określonego następnika.

Zalecenia

1. Jeżeli istnieją tylko dwie możliwości, to używaj instrukcji if.
2. Jeżeli decyzja wyboru nie bazuje na wartości jednego wyrażenia dyskretnego, to używaj instrukcji if.
3. Jeżeli jest wiele możliwości bazujących na wartości jednego wyrażenia dyskretnego, używaj instrukcji case.
4. Jeżeli istnieje kilka możliwości bazujących na wartości jednego wyrażenia dyskretnego używaj instrukcji case, albo if tak, aby uzyskać bardziej czytelny program.
5. Staraj się nie używać stylu „mieszanego”

```
k : Integer;
...
case k is
    when 1 => ...
    when 7 => ...
    when -2 | 3 => ...
    when others =>
        if k mod 2 = 0 then
            ...
        else
```

```
11                case k is
12                    when 11 => ...
13                    ...
14                end case;
15            end if;
16 end case;
```

6. Instrukcję wyboru należy stosować tylko wtedy, gdy podzbiory wartości określone listami wyboru ściśle do siebie przylegają. Jeżeli tak nie jest, to często lepiej zastosować instrukcję if.

7. Część others powinna być wykonywana w przypadkach szczególnych tzn. takich które rzadko mogą wystąpić.

4.7 Pętle

Instrukcja pętli jest następująco zdefiniowana w notacji EBNF:

```
loop_statement ::=
    [loop_simple_name;]
    loop
        sequence_of_statements;
    end loop [loop_simple_name];
```

Często algorytmy zawierają fragmenty wymagające powtarzania pewnej instrukcji lub ciągu instrukcji. Fragment algorytmu realizujący powtarzanie ciągu instrukcji nazywamy *pętlą*, albo *iteracją*. Proste algorytmy iteracyjne prezentowane są na lekcjach matematyki w szkole średniej, a ich typowym przykładem jest obliczanie kolejnych wyrazów pewnego ciągu (postęp arytmetyczny, albo postęp geometryczny). Najprostszą formą zapisu algorytmu iteracyjnego w Adzie jest instrukcja pętli postaci

```
loop --- początek pętli
   Ciag_Instrukcji;
end loop; -- koniec pętli
```

Po wejściu do pętli wykonywany jest Ciag_Instrukcji i jest on wykonywany w nieskończoność, chyba że jedna z instrukcji ciągu zapewnia zakończenie iteracji. Natychmiast powstaje pytanie czy pętle nieskończone mają sens i bardziej ogólne – czy wszystkie programy muszą się kończyć. W naszym wstępnym kursie programowania można powiedzieć, że tak, ale można sobie wyobrazić programy, które nie muszą się kończyć. Przykładem takiego programu może być program monitorujący lub sterujący pewnym ciągłym procesem produkcyjnym np. procesem wydobywczym w kopalni odkrywkowej. Proces taki odbywa się praktycznie w „nieskończoność" ponieważ kopalnia pracuje cały czas i urządzenia wydobywcze są monitorowane 24 godziny na dobę. Oczywiście pojęcie nieskończoności nie jest w tym przypadku tym samym co nieskończoność w matematyce. Doskonale wiemy, że moment wyczerpania się złoża wyznacza czas życia kopalni, a poza tym komputer można wyłączyć i wtedy program przestanie działać. Sam program może być jednak zaprojektowany tak, że działa ciągle, przy czym jest oczywiste, że sposób jego działania powinien dostosowywać się

do zmian zachodzących w otoczeniu, z którym komunikuje się komputer na którym program działa.

Innym przykładem procesu nieskończonego jest pętla wyrażająca nieosiągalną nieśmiertelność (Barnes 1998):

```
loop
   Pracuj;
   Jedz;
   Odpoczywaj;
end loop;
```


Bardziej konkretnych przykładów nieskończonych iteracji dostarcza matematyka, gdzie wiele pojęć określonych jest jako granice. Typowymi przykładami są ciągi i szeregi. Weźmy pod uwagę definicję liczby e, która jest podstawą logarytmu naturalnego (Żakowski i Decewicz 1997). Zgodnie z definicją mamy

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}, \text{ przy czym } k! = \begin{cases} 1, & k = 0 \\ k * (k-1), & k = 1, 2, \ldots \end{cases}$$

Algorytm realizujący obliczenie liczby e można zapisać w postaci:

```
1  declare
2     Liczba_E : Float := 0.0;
3     K : Natural := 0;
4     Skladnik : Float := 1.0;
5  begin
6     loop
7        K := K + 1;
8        Skladnik := Skladnik/Float(K);
9        Liczba_E := Liczba_E + Skladnik;
10    end loop;
```

łatwo zauważyć, że iteracja realizowana przez instrukcję pętli nie skończy się, co odpowiada matematycznej definicji liczby e. Ponieważ reprezentacja komputerowa liczb rzeczywistych jest skończona, po wykonaniu pewnej ilości instrukcji objętych pętlą zmienna Skladnik przyjmie wartość zero i dalsze obliczenia stracą sens. W praktyce zadowalamy się obliczeniem liczby e z pewną zadana dokładnością i przerywamy iterację, gdy osiągamy tę dokładność. Aby wyjść z pętli możemy wewnątrz niej wykonać instrukcję:

```
exit;
```

której opis w notacji EBNF wygląda następująco:

```
exit_statement ::=
    exit [loop_simple_name][when Condition];
```

Jeżeli zostaje napotkana instrukcja exit, to obliczenia są przerywane i sterowanie zostaje przekazane do punktu po end loop. W naszym przykładzie możemy przyjąć, że wykonujemy pętlę N razy i kończymy ją, gdy K = N. W tym przypadku nasza pętla ma postać:

```
1  loop
2     if K = N then
3        exit;
4     end if;
```

```
5      K := K + 1;
6      Skladnik := Skladnik/Float(K);
7      Liczba_E := Liczba_E + Skladnik;
8  end loop;
```

Musimy oczywiście przyjąć odpowiednią wartość dla N, jeżeli chcemy obliczyć zadowalające przybliżenie. Warunek wyjścia z pętli zapisany z użyciem instrukcji if jest tak często stosowany, że Ada ma specjalną, równoważną konstrukcję:

```
exit when Warunek;
```

w której Warunek jest wyrażeniem logicznym. Instrukcja wyjścia z pętli może wystąpić w dowolnym miejscu w jej wnętrzu[6], przy czym należy zwrócić uwagę na dwa szczególne położenia tej instrukcji: pierwsza instrukcja pętli i ostatnia instrukcja pętli.

Pierwszy przypadek – tzw. pętla z testem początkowym – występuje wtedy, gdy chcemy sprawdzić warunki wykonywania instrukcji wewnętrznych przed ich wykonaniem. Sytuacja ta jest tak często spotykana, że zamiast pisać

```
loop
   exit when Warunek;
   ...
   ...
end loop;
```

piszemy

```
while Warunek loop
   ...
   ...
end loop;
```

Pętlę tego rodzaju nazywamy pętlą, albo instrukcją while. Z instrukcją tą wiąże się wykonanie następujących akcji:

Wartościowanie wyrażenia Warunek, które przyjmujące wartości True albo False.

Jeżeli Warunek ma wartość True, to wykonywany jest ciąg instrukcji i powtarzany jest krok 1, a jeżeli Warunek ma wartość False, to kończone jest wykonywanie instrukcji while.

Wykorzystanie tej instrukcji ilustruje poniższy program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Odliczanie is
5      Licznik : Natural;
6  begin
7      Licznik := 10;
8      while Licznik > 0 loop
9          if Licznik = 3 then
10             Put ("Zaplon");
11             New_Line;
12         end if;
```

[6] Ogólnie pętlę tego typu nazywa się czasami pętlą z wartownikiem (ang. *sentinell*).

```
13          Put (Licznik, 3);
14          New_Line;
15          Licznik := Licznik - 1;
16          delay 1.0;
17       end loop;
18       Put ("Start");
19    end Odliczanie;
```

Zwróćmy uwagę na instrukcję delay 1.0;, która przerywa działanie programu na jedną sekundę.

W języku Ada **nie istnieje** specjalna wersja instrukcji pętli z testem końcowym (czyli takiej, w której instrukcja exit when Warunek jest ostatnią instrukcją pętli) podobnej do znanej z języka Pascal instrukcji repeat .. until Warunek (Teixeira i Pacheco 2002), czy z języka C i Java – do .. while (Warunek); (Eckel 2001, Stroustrup 1991, Kernighan i Ritchie 1987).

Drugim przykładem jest obliczenie sumy określonej wzorem $\sum_{k=1}^{N} k^2$. Zadanie obliczenia tej sumy rozwiązać można następująco:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Suma_Kwadratow is
5     N, K : Natural;
6     Suma : Natural := 0;
7  begin
8     Put ("N = ");
9     Get (N);
10    K := N;
11    New_Line;
12    while K > 0 loop
13       Suma := Suma + K*K;
14       K := K - 1;
15    end loop;
16    Put ("Suma kwadratow pierwszych ");
17    Put (N, 2);
18    Put (" liczb calkowitych dodatnich = ");
19    Put (Suma, 3);
20 end Suma_Kwadratow;
```

Przy używaniu instrukcji iteracyjnych, a w szczególności instrukcji while, należy zwrócić uwagę na następujące zagadnienia:

1. Wykonanie każdego ciągu instrukcji powinno powodować zbliżanie się do spełnienia warunku zakończenia instrukcji iteracyjnej. Oznacza to, że na warunek zakończenia powinny mieć wpływ powtarzane obliczenia.
2. Jeżeli warunek reprezentowany wyrażeniem logicznym nie jest początkowo spełniony, to instrukcja while jest pusta tzn. nie jest wykonywana żadna akcja po sprawdzeniu warunku.
3. Dobrym zwyczajem programistycznym jest sformułowanie pewnej zależności nazywanej *niezmiennikiem pętli*, która pozwala kontrolować stan iteracji.

4. Należy unikać powtarzania tych samych obliczeń. Oznacza to, że wewnątrz instrukcji iteracyjnych nie należy stosować wyrażeń, w których żadna ze zmiennych nie zmienia swojej wartości.

W celu ilustracji trzeciego zalecenia rozważmy zadanie obliczenia ilorazu Q dwóch liczb naturalnych $X \neq 0$ i $Y \neq 0$ oraz reszty R z tego dzielenia. Niech początkowo $Q = 0$ i $R = X$. Problem rozwiązujemy obliczając ile razy Y można odjąć od X. Nasze zadanie rozwiązuje program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Niezmiennik_Petli is
5     X, Y : Natural; -- X - dzielna, Y - dzielnik
6     Q, R : Natural; -- Q - iloraz, R - reszta
7  begin
8     Put ("Podaj dzielna : ");
9     Get (X);
10    Put ("Podaj dzielnik : ");
11    Get (Y);
12    New_Line;
13    Q := 0;
14    R := X;
15    if Y /= 0 then
16       Put ("Niezmiennik przed wejściem do pętli");
17       New_Line;
18       Put (Q*Y + R, 2);
19       New_Line;
20       Put ("Niezmiennik podczas wykonywania pętli");
21       New_Line;
22       while R >= Y loop
23          R := R - Y;
24          Q := Q + 1;
25          Put (Q*Y + R, 2);
26          New_Line;
27       end loop;
28       Put ("Dzielna = ");
29       Put (X, 2);
30       Put (" dzielnik = ");
31       Put(Y,2);
32       New_Line;
33       Put ("Iloraz = ");
34       Put (Q, 2);
35       Put (" reszta = ");
36       Put (R, 2);
37    else
38       New_Line;
39       Put ("Nie można dzielić przez zero");
40    end if;
41 end Niezmiennik_Petli;
```

Łatwo zauważyć, że równanie $X = Q * Y + R$ jest niezmiennikiem pętli, gdyż jest ono spełnione przy każdym rozpoczęciu ciągu instrukcji objętych instrukcją while. Stosowanie niezmiennika pętli jest przykładem wykorzystania reguł

tworzenia programów formalnie poprawnych. Tutaj sygnalizujemy jedynie to zagadnienie, natomiast zainteresowanych Czytelników odsyłamy do książki Alagića i Arbiba (Alagić i Arbib 1978).

Zgodnie z punktem 4. podanych zaleceń instrukcję:

```
while I < 3 * N loop
   Wektor (I) := X + Y + Z + Z * I;
   I := I + 1;
end loop;
```

należy zastąpić przez ciąg instrukcji postaci:

```
K := 3 * N;
U := X + Y + Z;
while I < K loop
   Wektor (I) := U + Z * I;
   I := I + 1;
end loop;
```

Jeżeli chcemy powtarzać wykonanie pewnego ciągu instrukcji i ilość powtórzeń nie zależy od wyników wykonania tego ciągu, to do realizacji takiego procesu iteracyjnego należy stosować instrukcję iteracyjną for. W celu ilustracji użycia tej formy pętli wróćmy do programu Suma_Kwadratow. Nietrudno widzieć, że w tym przykładzie ilość wykonań pętli jest z góry znana i dlatego pętla for jest dobrym rozwiązaniem.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Suma_Kwadratow_For is
5      N : Natural;
6      Suma : Natural := 0;
7  begin
8      Put ("N = ");
9      Get (N);
10     Suma := 0;
11     New_Line;
12     for K in 1..N loop
13         Suma := Suma + K*K;
14     end loop;
15     Put ("Suma kwadratow pierwszych ");
16     Put (N, 2);
17     Put (" liczb calkowitych dodatnich = ");
18     Put (Suma, 3);
19 end Suma_Kwadratow_For;
```

Zauważmy, że deklaracja zmiennej K nie została jawnie podana. Zmienna ta nazywana jest *zmienną sterującą*, albo *licznikiem* instrukcji for. W ogólnym przypadku zakres zmienności licznika określony jest przez Zakres_Dyskretny pętli, a więc pętlę tę zapisujemy w postaci:

```
for Licznik in Zakres_Dyskretny loop
   Ciag_Instrukcji;
end loop;
```

Typ zmiennej Licznik wynika z typu Zakres_Dyskretny i w naszym przykładzie jest to typ Natural. Ponieważ licznik jest deklarowany przez instrukcję for, przestaje on istnieć po wyjściu z pętli. Należy podkreślić, że wartości licznika nie można samowolnie zmieniać wewnątrz pętli, a zmiany jego wartości realizowane są automatycznie przez samą pętlę.

Ogólnie rzecz biorąc wykonanie instrukcji for .. loop związane jest z wykonaniem wszystkich jej instrukcji określoną ilość razy, ale trzeba pamiętać, że instrukcją zawartą w Ciag_Instrukcji może być m.in. instrukcja exit lub return, która kończy wykonanie tej wersji instrukcji loop **przed** wykonaniem wszystkich założonych iteracji.

Teraz wyjaśnimy czym jest Zakres_Dyskretny. Jest to nazwa podtypu dyskretnego z opcjonalnym podaniem zakresu, lub sam zakres. W podanym przykładzie mamy podany tylko zakres 1..N, przy czym ograniczenie górne zakresu jest zadeklarowane jako zmienna typu Natural, o nadanej wartości przed wejściem do pętli.

Jeżeli wewnątrz pętli zmienimy to ograniczenie, to nie będzie to miało wpływu na ilość iteracji. Wracając do naszego przykładu, jeżeli napiszemy:

```
N := 10;
for K in 1..N loop
   Suma := Suma + K * K;
   N := 20;
end loop;
```

to pętla zostanie wykonana 10 razy, a nie 20.

Ograniczenie dolne i górne zakresu mogą być wyrażeniami, a nie stałymi lub zmiennymi i wartości tych ograniczeń mogą być określone przy wejściu do pętli.

Z tego co dotychczas powiedzieliśmy, Zakres_Dyskretny może być dowolnego typu dyskretnego. Możemy więc napisać:

```
for Dzien in Poniedzialek .. Niedziela loop
   ...
   ...
end loop;
```

W tym przypadku Dzien jest typu Dzien_Tygodnia i w związku z tym możemy podaną instrukcję zastąpić przez równoważną instrukcję postaci:

```
for Dzien in Dzien_Tygodnia loop
   ...
   ...
end loop;
```

Jeżeli chcemy wykonywać pętlę tylko dla wszystkich dni roboczych, to piszemy:

```
for Dzien in Dzien_Tygodnia range Poniedziałek .. Piatek loop
   ...
   ...
end loop;
```

albo krócej i jaśniej:

```
for Dzien in Dzien_Roboczy loop
   ...
   ...
end loop;
```

Typowo Licznik zmienia swą wartość w porządku rosnącym. Porządek przeciwny może być również realizowany i wtedy piszemy:

```
for K in reverse 1..N loop
```

Wykorzystamy tę możliwość w nowej wersji programu Odliczanie.


```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Odliczanie_For is
5     N : Natural := 10;
6  begin
7     for K in reverse 1 .. N loop
8        if K = 3 then
9           Put ("Zaplon");
10          New_Line;
11       end if;
12       Put (K, 3);
13       New_Line;
14       delay 1.0;
15    end loop;
16    Put ("Start");
17 end Odliczanie_For;
```

Pętlę for można opuścić jeżeli jest spełniony pewien Warunek_Wyjscia. Realizujemy to umieszczając w ciągu instrukcji pętli instrukcję warunkową:

```
if Warunek_Wyjscia then
   Ostatnia_Wartosc_Licznika := K;
   exit;
end if;
```

przy czym zmienna Ostatnia_Wartosc_Licznika, zadeklarowana poza pętlą, służy do wyprowadzenia wartości licznika na zewnątrz pętli.

Pętle mogą być zagnieżdżone tzn. wewnątrz pętli może być następna pętla itd. Umieszczenie instrukcji powodującej wyjście z pętli wewnętrznej, kończy tę pętlę i dalej wykonywane są instrukcje pętli zewnętrznej, chyba że, spełniony jest warunek zakończenia tej pętli. Aby uniknąć kłopotliwego sprawdzania takich warunków, pętla zewnętrzna może mieć nazwę, do której można się odwołać jeżeli chcemy wyjść wykonując instrukcje pętli wewnętrznej. Można więc napisać:

```
Petla:
for I in 1..N loop
   for J in 1..M loop
      if Warunek_Wyjscia then
         Ostatnia_Wartosc_I := I;
         Ostatnia_Wartosc_J := J;
         exit Petla;
```

```
        end if;
    end loop;
end loop Petla;
```

Zwróćmy uwagę na to, że identyfikator pętli zewnętrznej występuje po słowie exit i po nawiasie zamykającym (end loop) tę petlę. Jeżeli stosujemy konstrukcję z nazwą (nie jest to etykieta), to instrukcję wyjścia możemy napisać w krótszej formie:

```
exit Petla when Warunek_Wyjscia;
```

Kończąc omawianie pętli podamy kilka zaleceń. Instrukcja loop jest uogólnioną formą instrukcji iteracyjnej. Teoretycznie każdą z instrukcji while i for można wyrazić jako instrukcję loop zawierającą jedną instrukcję exit. W praktyce prowadziłoby to do niejasnych programów. Należy więc używać instrukcji while i for tam gdzie jest to możliwe, a instrukcji loop tylko wtedy, gdy nie ma innej możliwości realizacji algorytmu iteracyjnego. Instrukcja loop jest wygodna, gdy zakończenie procesu iteracyjnego powodowane jest przez spełnienie jednego z kilku niezależnych warunków.

Zamykając ten rozdział podajemy dwa programy, które ilustrują zastosowanie omawianych instrukcji. Pierwszy program realizuje algorytm złotego podziału poszukiwania punktu minimalnego funkcji rzeczywistej, unimodalnej[7] w przedziale $[L, P] \subset \Re$.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4  with Ada.Numerics.Elementary_Functions;
5                                  use Ada.Numerics.Elementary_Functions;
6
7  procedure Zloty_Podzial is
8     -- Program oblicza minimum funkcji unimodalnej w przedziale [L,P]
9     -- metodą złotego podziału
10    Max_Iter : constant Positive := 100; -- dopusczalna ilość iteracji
11    Alfa : constant Float := (Sqrt (5.0) - 1.0) / 2.0;
12    Beta : constant Float := 1.0 - Alfa;
13                   -- współczynniki złotego podziału
14    Dlug_Min : Float; -- minimalna długość przedziału
15    Punkt_Min, Min : Float; -- wyniki poszukiwania minimum
16    Numer_Iter : Positive;
17    L, P : Float; -- końce dziedziny funkcji
18    La, Mi : Float; -- współczynniki lambda i mi
19    F_La, F_Mi : Float;
20    Ro : Float;
21
22    function Funkcja (X : in Float) return Float is
23    begin
24       return X*X + 2.0*X;
25    end Funkcja;
26
27    procedure Wypisz_Wyniki_Iteracji is
28    begin
```

[7] Na podstawie (Bazaraa i Shetty 1982, str. 270), (Polak 1971, str. 49).

```
29          Put (Numer_Iter, 2);
30          Put (" L = ");
31          Put (L, 2, 3, 0);
32          Put (" P = ");
33          Put (P, 2, 3, 0);
34          Put (" La = ");
35          Put (La, 2, 3, 0);
36          Put (" Mi = ");
37          Put (Mi, 2, 3, 0);
38          Put (" F_La = ");
39          Put (Funkcja(La), 2, 3, 0);
40          Put (" F_Mi = ");
41          Put (Funkcja(Mi), 2, 3, 0);
42      end Wypisz_Wyniki_Iteracji;
43
44  begin
45      loop
46          Put ("Podaj minimalna długość przedziału : ");
47          Get (Dlug_Min);
48          if Dlug_Min <= 0.0 then
49              Put (" Minimalna długość przedziału musi być dodatnia");
50              New_Line;
51          end if;
52          exit when Dlug_Min > 0.0;
53      end loop;
54      loop
55          Put ("Podaj lewy koniec przedziału : ");
56          Get (L);
57          Put ("Podaj prawy koniec przedziału : ");
58          Get (P);
59          if L >= P then
60              Put ("Lewy koniec musi być < od prawego");
61          end if;
62          exit when P > L + Dlug_Min;
63      end loop;
64      Numer_Iter := 1;
65      Ro := (P - L);
66      La := L + Beta*Ro;
67      Mi := L + Alfa*Ro;
68      F_La := Funkcja (La);
69      F_Mi := Funkcja (Mi);
70      Wypisz_Wyniki_Iteracji;
71      while (Ro >= Dlug_Min) or (Numer_Iter >= Max_Iter) loop
72          New_Line;
73          Numer_Iter := Numer_Iter + 1;
74          if F_La > F_Mi then
75              L := La;
76              Ro := P - L;
77              La := Mi;
78              Mi := L + Alfa*Ro;
79              F_La := F_Mi;
80              F_Mi := Funkcja (Mi);
81          else
82              P := Mi;
```

```
83              Ro := P − L;
84              Mi := La;
85              La := L + Beta∗Ro;
86              F_Mi := F_La;
87              F_La := Funkcja (La);
88          end if;
89          Wypisz_Wyniki_Iteracji;
90      end loop;
91      Punkt_Min:= L + Ro/2.0;
92      Min := Funkcja (Punkt_Min);
93      New_Line;
94      Put ("Punkt minimum = ");
95      Put (Punkt_Min, 2, 3, 0);
96      Put (" Minimum = ");
97      Put (Min, 2, 3, 0);
98  end Zloty_Podzial;
```

Wyniki otrzymane w przypadku, gdy minimalna długość przedziału w którym znajduje się punkt minimum wynosiła 1.0E−1 są następujące:

```
1 L = −3.000 P = 5.000 La = 0.056 Mi = 1.944 F_La = 0.115 F_Mi = 7.669
2 L = −3.000 P = 1.944 La = −1.111 Mi = 0.056 F_La = −0.988 F_Mi = 0.115
3 L = −3.000 P = 0.056 La = −1.833 Mi = −1.111 F_La = −0.306 F_Mi = −0.988
4 L = −1.833 P = 0.056 La = −1.111 Mi = −0.666 F_La = −0.988 F_Mi = −0.888
5 L = −1.833 P = −0.666 La = −1.387 Mi = −1.111 F_La = −0.850 F_Mi = −0.988
6 L = −1.387 P = −0.666 La = −1.111 Mi = −0.941 F_La = −0.988 F_Mi = −0.997
7 L = −1.111 P = −0.666 La = −0.941 Mi = −0.836 F_La = −0.997 F_Mi = −0.973
8 L = −1.111 P = −0.836 La = −1.006 Mi = −0.941 F_La = −1.000 F_Mi = −0.997
9 L = −1.111 P = −0.941 La = −1.046 Mi = −1.006 F_La = −0.998 F_Mi = −1.000
10 L = −1.046 P = −0.941 La = −1.006 Mi = −0.981 F_La = −1.000 F_Mi = −1.000
11 L = −1.046 P = −0.981 La = −1.022 Mi = −1.006 F_La = −1.000 F_Mi = −1.000
Punkt minimum = −1.014 Minimum = −1.000
```

łatwo sprawdzić, że rozwiązanie dokładne jest równe:

Punkt minimum = −1.0 Minimum = −1.0

Drugi przykład jest uogólnieniem problemu obliczenia przybliżenia liczby e i służy do tablicowania wartości funkcji Exp w przedziale $[Lewy, Prawy] \subset \Re$, przy zadanej ilości równoodległych punktów tego przedziału, w których liczymy wartości funkcji.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4  with Ada.Numerics.Elementary_Functions;
5                                   use Ada.Numerics.Elementary_Functions;
6
7  procedure Funkcja Exp is
8      Epsilon : Float := 1.0e−6;
9      Argument : Float;
10     Przyrost : Float;
11     Lewy : Float;
12     Prawy : Float;
13     Ile_Punktow : Positive;
14     Max_Iter : constant Positive := 100;
15
16     function Wykladnicza (X : in Float;
```

```
17                              Eps : in Float;
18                              Max : Positive ) return Float is
19          Suma : Float;
20          Suma_Nastepna : Float;
21          K : Positive;
22          Wyraz : Float;
23      begin
24          Suma := 1.0;
25          K := 1;
26          Wyraz := X/Float(K);
27          Suma_Nastepna := Suma + Wyraz;
28          while abs (Suma_Nastepna - Suma) > Eps or K > Max) loop
29              Suma := Suma_Nastepna;
30              K := K + 1;
31              Wyraz := Wyraz*X/Float(K);
32              Suma_Nastepna := Suma + Wyraz;
33          end loop;
34          return Suma_Nastepna;
35      end Wykladnicza;
36
37  begin
38      loop
39          Put ("Podaj lewy koniec przedzialu : ");
40          Get (Lewy);
41          Put ("Podaj prawy koniec przedzialu : ");
42          Get (Prawy);
43          if Lewy >= Prawy then
44              Put ("Lewy koniec musi byc < od prawego");
45          end if;
46          exit when Prawy > Lewy;
47      end loop;
48      Put ("W ilu punktach liczymy : ")
49      Get (Ile_Punktow);
50      Przyrost := (Prawy - Lewy)/Float(Ile_Punktow);
51      for I in 0..Ile_Punktow loop
52          Argument := Float(I)*Przyrost;
53          Put ("X = ");
54          Put (Argument, 3, 6, 0);
55          Put (" Wykladnicza(X) = ");
56          Put (Wykladnicza (Argument, Epsilon, Max_Iter), 3, 6, 0);
57          Put (" Exp(X) = ");
58          Put (Exp(Argument), 3, 6, 0);
59          New_Line;
60      end loop;
61  end Funkcja_Exp;
```

W celu sprawdzenia czy funkcja Wykladnicza dobrze liczy, program wypisuje wartości wyznaczane przez funkcję Exp z pakietu Ada.Numerics.Elementary_Functions. Wyniki uzyskane przez nasz program dla danych Lewy = 0.0, Prawy = 1.0 i Ile_Punktow = 10 są następujące:

X = 0.000000 Wykladnicza(X) = 1.000000 Exp(X) = 1.000000
X = 0.100000 Wykladnicza(X) = 1.105171 Exp(X) = 1.105171
X = 0.200000 Wykladnicza(X) = 1.221403 Exp(X) = 1.221403
X = 0.300000 Wykladnicza(X) = 1.349859 Exp(X) = 1.349859

```
X = 0.400000 Wykladnicza(X) = 1.491825 Exp(X) = 1.491825
X = 0.500000 Wykladnicza(X) = 1.648721 Exp(X) = 1.648721
X = 0.600000 Wykladnicza(X) = 1.822119 Exp(X) = 1.822119
X = 0.700000 Wykladnicza(X) = 2.013753 Exp(X) = 2.013753
X = 0.800000 Wykladnicza(X) = 2.225541 Exp(X) = 2.225541
X = 0.900000 Wykladnicza(X) = 2.459603 Exp(X) = 2.459603
X = 1.000000 Wykladnicza(X) = 2.718282 Exp(X) = 2.718282
```

Widzimy, że obydwie funkcje Wykladnicza i Exp dały te same wyniki. Nie należy tu pochopnie wnioskować, że zalecamy stosowanie własnych procedur i funkcji zamiast podprogramów bibliotecznych. Wręcz przeciwnie, jeżeli możemy skorzystać z podprogramów bibliotecznych, to należy tak zrobić. Celem naszego ostatniego przykładu była jedynie ilustracja jak stosować instrukcje iteracyjne i warunkowe.

Rozdział 5

Strukturalne typy danych

Dane w każdym programie tworzą abstrakcyjny opis rozwiązywanego problemu – tworzą jego model. Aby taki model był funkcjonalnie spójny, poszczególne dane muszą reprezentować wybrane cechy modelu. Jeżeli model ma strukturę prostą tzn. wszystkie jego cechy w dowolnej chwili dadzą się opisać informacjami prostymi, wtedy poznane już typy skalarne są wystarczającym narzędziem do konstrukcji modelu. W przeciwnym przypadku dogodnie jest patrzeć na model w sposób strukturalny tzn. najpierw widzimy model jako pewną całość, co stanowi pierwszy poziom szczegółowości, a następnie schodząc na niższy poziom mamy dane modelu, które mogą mieć pewną strukturę, a wartości tych danych nie reprezentują pojedynczych informacji, a raczej ich zespoły. Aby było możliwe takie spojrzenie na model musimy wprowadzić pojęcie typów strukturalnych, a co za tym idzie danych strukturalnych, które są niezbędne do budowy złożonych, ale za to systematycznie i czytelnie skonstruowanych modeli.

Podsumowując można wyróżnić dwie przyczyny uzasadniające korzystanie ze strukturalnych typów danych.

Po pierwsze, przy konstruowaniu oprogramowania metodą zstępującą *(ang. top-down design)* programista musi mieć możliwość opisywania danych na różnych poziomach złożoności. Na przykład dana reprezentująca datę obejmuje następujące składniki liczbowe: rok, miesiąc i dzień. Na wyższym poziomie dana taka jest traktowana jako całość, natomiast w podprogramie aktualizacyjnym datę należy wykonywać operację na składowych, czyli na liczbach reprezentujących odpowiednio rok, miesiąc i dzień. W ten sposób początkowa struktura danych opisana jest przy pomocy danych prostych, z których na wyższym poziomie tworzy się bardziej złożoną strukturę np. dane: rok, miesiąc, dzień łączone są w datę.

Po drugie, dane muszą być tak skonstruowane, aby operacje na danych strukturalnych można było traktować jako złożenie operacji na ich składowych. Jeżeli nie ma strukturalizacji danych, to każdemu prostemu obiektowi musi być przyporządkowana nazwa umożliwiająca jednoznaczne odwołanie się

do tego obiektu. W sytuacji, gdy ta sama operacja musi być zastosowana do grupy jednakowych obiektów danych, to dla przekształcenia każdego, pojedynczego obiektu, należy określić osobny operator. Przy grupowaniu obiektów danych w struktury i odwoływaniu się do elementów struktury przy pomocy mechanizmu indeksowania elementów, można korzystać z jednego operatora działającego w pętli dla wykonania potrzebnych operacji.

W Adzie wyróżnia się następujące strukturalne typy danych: tablicowe, rekordowe, chronione i zadaniowe, przy czym w tym rozdziale omawiamy jedynie pierwsze dwa. Typy chronione i monitory omawiamy w rozdziale 14, a kontynuacja omawiania typów rekordowego – typy klasowe związane z programowaniem obiektowym – w rozdziale 12.

5

5.1 Typy tablicowe

Tablice są najbardziej znanym mechanizmem strukturalizacji danych i są dostępne praktycznie we wszystkich językach programowania wysokiego poziomu. Ogólnie, *tablica jest skończonym ciągiem danych tego samego typu*. Dane tworzące tablicę nazywamy *elementami tablicy*. Bardziej formalnie konstrukcję tablicy można opisać następująco: niech $\mathcal{A}$ będzie pewnym zbiorem, z którego tworzymy zbiór $\mathcal{A}^n$; n – liczba naturalna, którego elementami są ciągi skończone postaci $A = A(i), i = 1, \ldots, n,\ A_i \in \mathcal{A}$.

Każdy tak skonstruowany ciąg nazywamy tablicą elementów zbioru $\mathcal{A}$, $A(i)$ nazywamy i -tym elementem, albo i -tą składową, a liczby i nazywamy indeksami elementów tej tablicy.

Łatwo zauważyć, że A jest nazwą tablicy, czyli całego ciągu elementów, a do wybrania konkretnego elementu tablicy służy *mechanizm indeksowania*.

Jeżeli chcemy zadeklarować zmienną M, która jest tablicą składającą się z dziesięciu liczb rzeczywistych, to możemy to zrobić przy pomocy deklaracji

```
M : array Positive range 1..10 of Float;
```

Jeżeli chcemy odwołać się do i – tego elementu tej tablicy, piszemy M(i), przy czym wartość indeksu «i» musi być odpowiedniego typu, oraz musi należeć do zakresu podanego w deklaracji tablicy. Jeżeli wartość indeksu nie spełnia ograniczeń, zostanie zgłoszony wyjątek CONSTRAINT_ERROR.

Pamiętamy, że w Adzie każda zmienna ma swój typ. W naszej deklaracji tablicy M jej typ nie ma identyfikatora, ale jest określony przy pomocy *opisu typu*. W takim przypadku mówimy o *typie anonimowym*. Trzeba jednak podkreślić, że zasady ścisłej typizacji zawsze obowiązują i jeżeli mamy deklaracje:

```
M : array Positive range 1..10 of Float;
A : array Positive range 1..10 of Float;
```

to instrukcja podstawienia A := M jest błędna ponieważ zmienne A i M są różnych typów. Nawet wtedy, gdy napiszemy:

```
A, M : array (Positive range 1..10)of Float;
```

to podana instrukcja podstawienia jest błędna ponieważ ostatnia deklaracja jest skróconym zapisem poprzedniej. Sytuacja nie jest jednak tak zła jak może się wydawać, ponieważ typy tablicowe można deklarować w zwykły sposób przypisując im nazwy. Wrócimy do tego zagadnienia dalej.

Deklaracje, które omawialiśmy określały tablice jednowymiarowe. W wielu zastosowaniach spotykamy tablice o większej ilości wymiarów i Ada umożliwia konstrukcję takich zespołów danych. Ogólnie możemy napisać

```
M : array (Typ_Indeksu_1 range Zakres_1,
           ...,
           Typ_Indeksu_N range Zakres_N) of Typ_Elementu;
```

Nie jest to formalnie poprawna deklaracja, ale oddaje istotę deklarowania tablicy N - wymiarowej, której elementami są dane typu Typ_Elementu, przy czym w Adzie ten typ może być dowolnym typem, a w szczególności typem strukturalnym. Zgodnie z tym, macierz prostokątną A wymiaru $N \times M$ możemy zadeklarować następująco:

```
N : Integer := ..; -- Można tu nadać wartość początkową
M : Integer := ..;
Macierz_N_Na_M : array (1..N, 1..M) of Float;
```

Zwróćmy uwagę na kilka nowych cech tej deklaracji. Zmienne N, M określające wymiary macierzy są typu podanego wcześniej przy ich deklaracji i ich wartości muszą być znane dopiero w momencie, gdy tworzona jest deklarowana zmienna typu tablicowego. Musimy więc wcześniej obliczyć wartości tych zmiennych, ale deklaracja tablicy może być dynamiczna, dzięki czemu utworzona zostanie struktura o potrzebnych wymiarach. Ten sposób deklarowania tablic jest bardzo dogodny, a Czytelnik znający język Pascal zauważy, że w Adzie mamy o wiele lepszy mechanizm deklarowania tablic. Co więcej, w Adzie możemy tworzyć typy tablicowe bez określania zakresów indeksów, co umożliwia pisanie ogólnych pakietów bibliotecznych zawierających np. realizacje numeryczne algorytmów algebry liniowej.

Podane dotąd deklaracje zmiennych tablicowych zawierały indeksy typu całkowitego. Ada nie narzuca takiego ograniczenia. Typ indeksu może być podtypem dowolnego typu dyskretnego. Możemy w związku z tym pisać:

```
Godziny_Pracy : array (Dzien_Tygodnia) of Float;
Stan_Nand : array (Boolean, Boolean) of Boolean;
```

Do typów tablicowych można stosować kilka atrybutów związanych z indeksami. Jeżeli mamy deklarację:

```
A : array (Typ_Indeksu range Pierwszy..Ostatni) of Typ;
```

to możemy napisać A'First i A'Last i otrzymamy odpowiednio wartości Pierwszy i Ostatni. Atrybut A'Length oblicza długość zakresu, czyli ilość wartości indeksu. Z kolei atrybut A'Range jest skróconą formą zapisu A'First .. A'Last. Poniższy program ilustruje wykorzystanie wymienionych atrybutów. W przypadku tablic wielowymiarowych, a taka występuje w programie przykładowym, podaje się jako parametr atrybutu numer indeksu, którego atrybut chcemy otrzymać.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Atrybuty_Tablic is
6     M : array (Positive range 1..3,
7                Positive range 1..2) of Float := ((1.0, 2.0),
8                                                  (3.0, 4.0),
9                                                  (5.0, 6.0));
10    procedure Wypisz_Element (I,J : in Positive;
11                              X : in Float;
12                              Nazwa : in String) is
13    begin
14       Put (Nazwa);Put ("(");Put (I, 2);
15       Put (",");Put (J, 2);Put (")␣=␣");
16       Put (X, 3, 5, 0);
17    end Wypisz_Element;
18
19 begin
20    Put ("Macierz:␣");
21    for Indeks_Pierwszy in M'range(1) loop
22       New_Line;
23       for Indeks_Drugi in M'range(2) loop
24          Wypisz_Element (Indeks_Pierwszy, Indeks_Drugi,
25                          M (Indeks_Pierwszy,
26                             Indeks_Drugi),
27                          "␣M␣");
28       end loop;
29    end loop;
30    New_Line;
31    Put ("Atrybuty␣macierzy:␣");
32    New_Line;
33    Put ("Wartosc␣dolna,␣gorna␣i␣ilosc␣wartosci␣pierwszego␣indeksu");
34    New_Line;
35    Put (M'First(1), 2);
36    Put ("␣␣");
37    Put (M'Last(1), 2);
38    Put ("␣␣");
39    Put (M'Length(1), 2);
40    New_Line;
41    Put ("Wartosc␣dolna,␣gorna␣i␣ilosc␣wartosci␣drugiego␣indeksu");
42    New_Line;
43    Put (M'First(2), 2);
44    Put ("␣␣");
45    Put (M'Last(2), 2);
46    Put ("␣␣");
47    Put (M'Length(2), 2);
48 end Atrybuty_Tablic;
```

Wyniki działania programu są następujące:

```
Macierz:
 M ( 1, 1) = 1.00000 M ( 1, 2) = 2.00000
 M ( 2, 1) = 3.00000 M ( 2, 2) = 4.00000
 M ( 3, 1) = 5.00000 M ( 3, 2) = 6.00000
```

```
Atrybuty macierzy:
Wartosc dolna, gorna i ilosc wartosci pierwszego indeksu
 1 3 3
Wartosc dolna, gorna i ilosc wartosci drugiego indeksu
 1 2 2
```

Należy zwrócić uwagę na sposób wykorzystania atrybutu 'Range, który został użyty w pętlach loop w miejsce zapisu:

```
for Indeks_Pierwszy in range 1..3 loop
   for Indeks_Drugi in range 1..2 loop
```

Takie korzystanie z tego atrybutu zalecamy, gdyż zwiększa ono przejrzystość programów i ich ogólność.

W podanym programie, zmiennej M nadano wartość początkową przy deklaracji tej zmiennej, przy czym zwracamy uwagę Czytelnika na stosowany zapis. Jeżeli możemy zmiennym typów tablicowych nadawać wartości początkowe, to powinien istnieć również mechanizm tworzenia stałych tych typów. Deklaracja stałej typu tablicowego jest podobna do deklaracji stałej typu skalarnego. Możemy napisać

```
Jutro : constant array (Dzien_Tygodnia) of Dzien_Tygodnia
          := (Wtorek, Sroda, Czwartek, Piatek, Sobota, Niedziela, Poniedziałek);
```

Jeżeli Dzien : Dzien_Tygodnia, to składowa Jutro(Dzien) jest dniem jutrzejszym.

Większość dotychczasowych rozważań na temat stałych i zmiennych typów tablicowych dotyczyła deklaracji stałych i zmiennych, których typ był anonimowy, a więc nie miał nazwy. Możliwość deklarowania typów anonimowych jest wyjątkowa i dotyczy omawianych tu tablic oraz zadań i obiektów chronionych. W Adzie nie musimy jednak deklarować zmiennych tablicowych przy pomocy opisu ich typu. Możemy i w większości zastosowań powinniśmy deklarować typy tablicowe, które otrzymują identyfikatory przy deklaracji. W związku z tym, nasza pierwsza deklaracja zmiennej tablicowej może być przekształcona do postaci:

```
type Wektor_10 is array (Positive range 1..10) of Float;
M : Wektor_10;
```

Teraz typ zmiennej M nie jest anonimowy i w przypadku, gdy mamy zmienną

```
A : Wektor_10;
```

to poprzednio błędna instrukcja przypisania A := M; staje się poprawna.

Omówione dotychczas sposoby deklarowania tablic mają jednak istotną wadę, ponieważ umożliwiają deklarowanie tablic jedynie o ustalonym zakresie indeksów. Ada zawiera mechanizm deklarowania typów tablicowych o nieokreślonych zakresach indeksów, co umożliwia pisanie podprogramów, których parametrami aktualnymi mogą być tablice o dowolnych wymiarach nie tracąc przy tym możliwości sprawdzania poprawności odwołań do elementów takich tablic. Tablice o nieokreślonych zakresach indeksów będziemy czasami nazywać *tablicami otwartymi*, a sposób ich definicji wyjaśnia następująca deklaracja:

```
type Wektor is array (Integer range <>)of Float;
```

Z deklaracji tej wynika, że Wektor jest typem tablicowym obejmującym tablice jednowymiarowe o elementach typu Float indeksowanych liczbami całkowitymi. Zapis range <> należy rozumieć w ten sposób, że w dalszej części programu nastąpi konkretyzacja zakresu. Konkretyzacja ta może przyjmować różne formy. Na przykład deklarujemy podtyp i zmienną:

```
subtype Wektor_10 is Wektor (1..10);
W : Wektor_10;
```

albo krócej:

```
W : Wektor (1..10);
```

Przy konkretyzacji zakresów, ich ograniczenia nie muszą być statyczne i mogą być wyrażeniami odpowiedniego typu dyskretnego podanego w deklaracji typu tablicowego o nieokreślonych zakresach.

5

Poniżej podano program ilustrujący wykorzystanie omówionych tablic otwartych. Tak jak wspomniano, procedury czytania i pisania zmiennych typów tablicowych, jako parametry formalne mają tablice otwarte, natomiast przy wywołaniu tych procedur, parametrami aktualnymi są konkretne tablice (określone w bloku), których ograniczenia zakresów są danymi wpisanymi przez użytkownika programu.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Typy_Tablicowe is
6    Ilosc_Wierszy : Positive;
7    Ilosc_Kolumn : Positive;
8    Ilosc_Skladowych : Positive;
9    type Macierz is array (Positive range <>, Positive range
10     <>) of Float;
11   type Wektor is array (Positive range <>) of Float;
12   procedure Czytaj_Macierz (X : out Macierz ) is
13   begin
14     for I in X'range(1) loop
15       for J in X'range(2) loop
16         Get (X(I,J));
17       end loop;
18     end loop;
19   end Czytaj_Macierz;
20   procedure Czytaj_Wektor (X : out Wektor ) is
21   begin
22     for I in X'range loop
23       Get (X(I));
24     end loop;
25     New_Line;
26   end Czytaj_Wektor;
27   procedure Wypisz_Macierz (X : in Macierz ) is
28   begin
29     for I in X'range(1) loop
30       for J in X'range(2) loop
31         Put (X(I,J));
```

```
32          end loop;
33          New_Line;
34       end loop;
35    end Wypisz_Macierz;
36    procedure Wypisz_Wektor (X : in Wektor ) is
37    begin
38       for I in X'range loop
39          Put (X(I));
40          New_Line;
41       end loop;
42    end Wypisz_Wektor;
43 begin
44    Put ("Podaj ilosc wierszy macierzy ");
45    Get (Ilosc_Wierszy);
46    Put ("Podaj ilosc kolumn macierzy ");
47    Get (Ilosc_Kolumn);
48    Put ("Podaj ilosc skladoych wektora ");
49    Get (Ilosc_Skladowych);
50    declare
51       M : Macierz (1 .. Ilosc_Wierszy, 1 .. Ilosc_Kolumn);
52       W : Wektor (1 .. Ilosc_Skladowych);
53    begin
54       Put ("Wpisz elementy macierzy prostokatnej ");
55       New_Line;
56       Czytaj_Macierz (M);
57       Put ("Oto wpisana macierz: ");
58       New_Line;
59       Wypisz_Macierz (M);
60       Put ("Wpisz elementy wektora ");
61       New_Line;
62       Czytaj_Wektor (W);
63       Put ("Oto wpisany wektor: ");
64       New_Line;
65       Wypisz_Wektor (W);
66    end;
67 end Typy_Tablicowe;
```

Wyniki działania programu dla danych Ilosc_Wierszy = 2, Ilosc_Kolumn = 3, Ilosc_Skladowych = 4 są następujace:

```
Oto wpisana macierz:
1.00000E+00 2.00000E+00 3.00000E+00
4.00000E+00 5.00000E+00 6.50000E+01
Oto wpisany wektor:
1.00000E+00
2.00000E+00
3.00000E+00
4.00000E+00
```

Jak pamiętamy, typ danych to nie tylko zbiór wartości, ale również podstawowe operacje jakie mogą być wykonywane na danych tego typu. Typy tablicowe nie są wyjątkiem, ale ilość operacji jakie mamy do dyspozycji bez wprowadzania własnych lub bibliotecznych podprogramów jest bardzo ograniczona. Dokładniej, znamy już instrukcję przypisania i mamy poza tym dwie relacje równości

i nierówności. W przypadku instrukcji przypisania (podstawienia) obowiązują następujące reguły: tablica której wartość przypisujemy i tablica która przyjmuje podstawianą wartość muszą być tego samego typu, a zakresy mogą być uzgodnione. Druga reguła sprowadza się do tego, że ilości odpowiednich indeksów muszą być takie same. Możemy napisać:

```
W : Wektor (1 .. 5);
V : Wektor (3 .. 7);
...
...
V := W;
```

Relacja równości dwóch tablic jest spełniona jeżeli są tego samego typu, mają zgodne zakresy (a nie tylko równą liczbę elementów!) i odpowiednie elementy są równe. Jeżeli którykolwiek z tych warunków nie jest spełniony, tablice są różne. Warto jednak wspomnieć, że jeżeli dwie tablice mają różne wymiary, to relacja równości ma wartość False, natomiast próba wykonania instrukcji podstawienia skończy się zgłoszeniem wyjątku CONSTRAINT_ERROR (zob. też rozdział 10).

W tym miejscu można wprowadzić pojęcie podtypu określonego i nieokreślonego. Typ określony jest podtypem przy użyciu którego możemy zadeklarować obiekt bez potrzeby jawnego określania zakresu, albo wartości początkowej. Tak więc Wektor_10 jest podtypem określonym, a Wektor jest typem nieokreślonym. Przy okazji omawiania tych pojęć dodajemy, że atrybuty 'First, 'Last, 'Length i 'Range mogą być stosowane do typów i podtypów tablicowych, ale tylko określonych[1].

W programie Atrybuty_Tablic zmiennej M nadaliśmy wartość początkową przy deklaracji zmiennej pisząc:

```
M : array (Positive range 1..3, Positive range 1..2) of Float
        := ((1.0, 2.0),
            (3.0, 4.0),
            (5.0, 6.0));
```

Taki układ liczb przypisywanych odpowiednim elementom tablicy nazywamy *pozycyjnym agregatem tablicowym*. Możemy jednak stosować agregaty innego rodzaju, nazywane *agregatami nazywanymi*. W tego rodzaju agregatach wartości przypisywane elementom tablicy poprzedzane są dwuznakowym symbolem =>. W przypadku tablicy M możemy napisać:

```
M : array (Positive range 1..3, Positive range 1..2) of Float
        := (1 => (1.0, 2.0),
            2 => (3.0, 4.0),
            3 => (5.0, 6.0));
```

Reguły tworzenia takich agregatów są podobne do reguł jakie obowiązują przy tworzeniu alternatyw w instrukcji wyboru. W celu ilustracji konstrukcji stosowanych przy tworzeniu agregatów nazywanych weźmy pod uwagę zmienną:

```
W : array (1 .. 10) of Float;
```

Możemy, w zależności od potrzeb pisać:

[1] Ale jeżeli zmienna typu Wektor jest parametrem podprogramu, to w tym podprogramie można używać wspomnianych atrybutów, ponieważ są one użyte zawsze dla *konkretnego* parametru.

```
W : array (1 .. 10) of Float := (1 .. 10 => 0.0);
W : array (1 .. 10) of Float := (9 | 10 => 1.0, 1 .. 8 => 0.0);
W : array (1 .. 10) of Float := (1 .. 8 => 0.0, others => 0.0);
```

W pierwszym przypadku wszystkie elementy otrzymują wartość 0.0, w drugim zapisie pierwszych osiem składowych ma wartość 0.0, a 9 i 10 wartość 1.0, natomiast ostatni przypadek, równoważny drugiemu, zawiera część others, która, jeżeli występuje, musi pojawić się na końcu.

Weźmy teraz pod uwagę znany już przypadek typu nieokreślonego i deklaracje:

```
type Wektor is array (Integer range <>) of Float;
W : Wektor (1 .. 5) := (3 .. 5 => 1.0, 6 | 7 => 0.0);
```

Zmiennej W przypisano wartość przy pomocy agregatu nazywanego, przy czym z zapisu wynikają ograniczenia tego agregatu, które wynoszą odpowiednio 3 i 7. Podstawienie agregatu do zmiennej powoduje tzw. *przesuwanie* (ang. sliding). Polega to na tym, że ograniczenia indeksów tablicy i agregatu są uzgadniane i składowe W(1)..W(3)= 1.0, a W(4), W(5)= 0.0. Jeżeli chcemy skorzystać z agregatu zawierającego część others, to możemy napisać:

```
W : Wektor (1 .. 5) := (3 .. 5 => 1.0, others => 0.0);
```

co daje inne wartości elementów wektora W(3)..W(5)= 1.0 i W(1),W(2)= 0.0. Jeżeli chcemy mieć te same wartości, które uzyskaliśmy przy pomocy pierwszego agregatu, to musimy napisać:

```
W : Wektor (1 .. 5) := (1.0, 1.0, 1.0, others => 0.0);
```

albo

```
W : Wektor (1 .. 5) := (1 .. 3 => 1.0, others => 0.0);
```

Mamy kilka zasad ogólnych:

Agregaty zawierające część others nie podlegają przesuwaniu.

Zakresy i wartości przed znakiem złożonym => muszą być statyczne, ale jest wyjątek: jeżeli jest tylko jedna możliwość składająca się z jednego wyboru, to może być określona dynamicznie. Ilustruje to fragment programu Agregaty_Tablicowe:

```
...
...
declare
    M : Macierz (1 .. Ilosc_Wierszy, 1 .. Ilosc_Kolumn)
        := (1 .. Ilosc_Wierszy => (1 .. Ilosc_Kolumn => 0.0));
    W : Wektor (1 .. Ilosc_Skladowych) := (others => 0.0);
  begin
    .
    .
  end;
end Agregaty_Tablicowe;
```

Przykładowe wyniki działania tego programu są następujące:

```
Podaj ilosc wierszy macierzy 2
Podaj ilosc kolumn macierzy 3
Podaj ilosc skladoych wektora 1
Oto macierz:
 0.00000E+00 0.00000E+00 0.00000E+00
 0.00000E+00 0.00000E+00 0.00000E+00
Oto wektor:
 0.00000E+00
```

Nie wolno tworzyć agregatów zawierających jednocześnie część pozycyjną i nazywaną, natomiast w przypadku agregatów wielowymiarowych można stosować różne formy na różnych poziomach. W przypadku zmiennej

```
M : array (Positive range 1..3, Positive range 1..2)of Float :=
```

możemy pisać:

```
((1 => 1.0, 2 => 2.0),
 (1 => 3.0, 2 => 4.0),
 (1 => 5.0, 2 => 6.0));
```

albo:

```
(1 => (1 => 1.0, 2 => 2.0),
 2 => (1 => 3.0, 2 => 4.0),
 3 => ( 5.0, 6.0));
```

W przypadku agregatu jednoelementowego nie można stosować zapisu pozycyjnego i w tym przypadku musimy stosować agregat nazywany. Poprawny jest więc zapis:

```
W : Wektor (1 .. 1) := (1 => 100.0);,
```

a błędny:

```
W : Wektor (1 .. 1):= (100.0); --Błąd !!!
```

Na zakończenie zalecamy stosowanie agregatów nazywanych, a nie pozycyjnych, ponieważ te pierwsze zabezpieczają przed przypadkową zamianą wartości przypisywanych elementom tablic.

5.2 Napisy

Znaki poznaliśmy w podrozdziale 3.9, a tutaj omówimy krótko napisy, które tworzymy ze znaków. Tabela 5.1 zawiera zbiór znaków kodu Latin-1, w której podano numery znaków i ewentualnie ich wygląd. Pamiętamy, że nie wszystkie znaki kodu Latin-1 możemy zobrazować i te znaki, o numerach z zakresów 0..31 i 127..159, nazywamy znakami sterującymi (w tabeli wyróżniono je symbolem „■"). Ponieważ w pakiecie Standard mamy również zdefiniowany typ wyliczeniowy Wide_Character, którego pierwszych 256 znaków to znaki identyczne ze znakami typu Character[2] nie możemy używać wyrażeń relacyjnych postaci 'X'< 'K' ponieważ powstaje niejednoznaczność, elementy którego z typów znakowych są

[2]Odpowiada to w pełni znakom zakodowanym w standardzie UNICODE (www.unicode.org 2000)

porównywane. Jeżeli chcemy porównać dwa elementy typu Character, to musimy kwalifikować jeden, albo obydwa argumenty. Piszemy więc poprawnie:

```
Character'('X')< 'K'
```

albo

```
Character'('X')< Character'('K')
```

W pakiecie Standard mamy również zdefiniowany wstępnie typ String, którego deklaracja jest następująca:

```
type String is array (Positive range <>) of Character;
```

Widzimy, że typ ten jest określony jako tablica otwarta znaków. Zasady używania zmiennych tego typu już znamy z poprzedniego podrozdziału, a tutaj wspomnimy o zdefiniowanych wstępnie operacjach jakie mogą być wykonywane na zmiennych napisowych. Napisy możemy porównywać i do tworzenia relacji z argumentami typu String używane są poznane poprzednio operatory relacyjne =, /=, < <=, > >=. Poza tym mamy dwuargumentową *operację sklejania*, czyli *konkatenacji*, której symbolem jest znak &. Przy pomocy tej operacji możemy łączyć dwa napisy, znak z napisem, napis ze znakiem i znak ze znakiem, przy czym wynikiem jest zawsze napis. Warto tu wspomnieć, że kolejny raz spotykamy się z pojeciem przeciążenia, tym razem funkcji, co oznacza, że to samo oznaczenie stosowane jest do formalnie różnych funkcji. Poniższy program ilustruje zastosowanie konkatenacji w przypadku różnych kombinacji argumentów:

```
1 with Ada.Text_Io;
2 use Ada.Text_Io;
3 procedure Napisy is
4 begin
5    Put ("Politechnika" & "␣Lodzka");
6    New_Line;
7    Put ("Wydzial" & '␣' & "F" & "T" & "I" & 'i' & "MS");
8 end Napisy;
```

Podany program wyprodukował następujące napisy:

```
Politechnika Lodzka
Wydzial FTIiMS
```

Na zakończenie tego podrozdziału dodajmy, że w pakiecie Standard mamy również zdefiniowany typ:

```
type Wide_String is array (Positive range <> of Wide_Character);
```

i analogiczne operatory relacyjne i operację konkatenacji.

Błędem jest jednak sądzić, że można „sklejać" wyłącznie napisy z napisami, i znakami. Operator ten może dotyczyć każdego innego typu danych – nawet rekordu czy tablicy, przy czym zarówno wynik, jak i argumenty nie muszą być typami ze sobą związanymi, np. poprawna jest deklaracja:

```
type a is record
   a, b, c : Integer;
end record;
```

```
type b is record
    q : Integer;
    w : Float;
end record;

function "&" (left : a; right : b) return Integer is
begin
    ...;
end "&";
```

Należy jednak pamietać, że operatory trzeba definiować **z sensem!** Interesującym i bardzo użytecznym przykładem zdefiniowania operatora & jest przykład wyjaśniony w rozdziale 13.4, do którego opisu czytelnik może zajrzeć już teraz mimo, że użyte są w nim bardziej złożone konstrukcje języka Ada.

Tabela 5.1: Zbiór znaków kodu Latin-1

0	■	32		64	@	96	‘	128	■	160		192	À	224	à
1	■	33	!	65	A	97	a	129	■	161	¡	193	Á	225	á
2	■	34	"	66	B	98	b	130	■	162	¢	194	Â	226	â
3	■	35	#	67	C	99	c	131	■	163	$	195	Ã	227	ã
4	■	36	$	68	D	100	d	132	■	164	¤	196	Ä	228	ä
5	■	37	%	69	E	101	e	133	...	165	¥	197	Å	229	å
6	■	38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
7	■	39	’	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	■	40	(	72	H	104	h	136	?	168	¨	200	È	232	è
9	■	41	)	73	I	105	i	137	‰	169	©	201	É	233	É
10	■	42	*	74	J	106	j	138	Š	170	ª	202	Â	234	â
11	■	43	+	75	K	107	k	139	‹	171	«	203	Ä	235	ä
12	■	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	■	45	-	77	M	109	m	141	■	173	-	205	Í	237	Í
14	■	46	.	78	N	110	n	142	■	174	®	206	Î	238	î
15	■	47	/	79	O	111	o	143	■	175	¯	207	Ï	239	ï
16	■	48	0	80	P	112	p	144	■	176	°	208	Đ	240	ð
17	■	49	1	81	Q	113	q	145	■	177	±	209	Ñ	241	ñ
18	■	50	2	82	R	114	r	146	■	178	²	210	Ò	242	ò
19	■	51	3	83	S	115	s	147	■	179	³	211	Ó	243	ó
20	■	52	4	84	T	116	t	148	■	180	´	212	Ô	244	ô
21	■	53	5	85	U	117	u	149	■	181	µ	213	Õ	245	õ
22	■	54	6	86	V	118	v	150	■	182	¶	214	Ö	246	ö
23	■	55	7	87	W	119	w	151	■	183	·	215	×	247	÷
24	■	56	8	88	X	120	x	152	■	184	¸	216	Ø	248	ø
25	■	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	■	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
27	■	59	;	91	[	123	{	155	›	187	»	219	Û	251	û
28	■	60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
29	■	61	=	93	]	125	}	157	■	189	½	221	Ý	253	ý
30	■	62	>	94	↑	126	~	158	■	190	¾	222	Þ	254	þ
31	■	63	?	95	_	127	■	159	Ÿ	191	¿	223	ß	255	ÿ

5.3 Rekordy

W odróżnieniu od tablic rekordy są strukturami danych, które tworzone są z danych różnych typów. Typowym przykładem jest struktura reprezentująca datę. Jak wiadomo, datę tworzą następujące dane: Rok, Miesiac i Dzien. Możemy przyjąć, że dana opisująca numer roku jest liczbą typu Positive z zakresu 1..2500, dana opisująca miesiąc może być podtypu utworzonego na bazie typu Positive i ograniczonego do przedziału 1..12, albo typu wyliczeniowego:

```
type Nazwy_Miesiecy is (Styczen, Luty, Marzec,
                        Kwiecien, Maj, Czerwiec,
                        Lipec, Sierpien, Wrzesien,
                        Pazdziernik, Listopad, Grudzien);
```

a dana reprezentująca dzień, podtypu okrojonego do przedziału 1..31. Data powinna więc należeć do typu utworzonego z połączenia trzech wymienionych podtypów. Połączenie takie uzyskuje się przy pomocy następującej deklaracji:

```
type Data is
   record
      Rok : Positive range 1 .. 2500;
      Miesiac : Positive range 1 .. 12;
      Dzien : Positive range 1 .. 31;
   end record;
```


Składowe rekordu nazywa się *polami rekordu*, a ich nazwy są nazywane *identyfikatorami pól*. Nie ma żadnych ograniczeń jeżeli chodzi o typy danych dla składowych rekordu. Jedynym warunkiem, który musi spełnić specyfikacja tego typu, jest to, że muszą być w niej użyte nazwy już znane.

Niech Dzien będzie zmienną typu rekordowego Data, czyli mamy deklarację:

```
Dzien : Data;
```

Pierwszym problemem jaki napotykamy jest odwoływanie się do poszczególnych składowych zmiennej. Składową zmiennej typu rekordowego wybieramy przez wypisanie identyfikatora zmiennej i nazwy składowej poprzedzonej kropką. W rozważanym przykładzie możemy napisać:

```
Dzien.Rok := 2000;
Dzien.Miesiac := 6;
Dzien.Dzien := 22;
```

Nazwy składowych zmiennej rekordowej nazywane są *selektorami rekordu* i używane są w programie dokładnie tak samo jak zmienne odpowiednich typów. Nie istnieją zdefiniowane wstępnie operacje umożliwiające wykonywanie działań na rekordach, a w szczególności nie istnieje relacja porządku dla rekordów. Wartość zmiennej rekordowej może być jednak podstawiona do innej zmiennej tego samego typu rekordowego. Podobnie jak zmiennym innych typów, zmiennym rekordowym można nadawać wartości początkowe przy deklaracji. Robimy to przy pomocy agregatów pozycyjnych, albo nazywanych. Kontynuując nasz przykład z datami możemy napisać:

```
Dzien : Data := (2000, 6, 22);
Inny_Dzien : Data;
Inny_Dzien := Dzien;
```

albo

```
Inny_Dzien := (Miesiac => 6, Dzien => 22, Rok => 2000);
```

Należy zwrócić uwagę na to, że w przypadku agregatu pozycyjnego kolejność nadawania wartości składowym musi być taka sama w jakiej występują odpowiednie deklaracje składowych w deklaracji typu rekordowego, natomiast

w przypadku agregatu nazywanego kolejność przypisań jest dowolna. Ponieważ ilość składowych rekordu jest zawsze znana, agregaty rekordowe są mniej skomplikowane od tablicowych. Warto jednak pamiętać o kilku zasadach.

- ⇝ W przypadku agregatów nazywanych nie wolno stosować zakresów, a znak pionowej kreski można używać do składowych tego samego typu bazowego. Podobnie, wybór others jest dozwolony tylko wtedy, gdy pozostałe składowe istnieją i są tego samego typu.
- ⇝ W niektórych przypadkach agregaty rekordowe mogą być zapisywane w formie mieszanej, ale jeżeli taką postać stosujemy, to zapis pozycyjny musi poprzedzać zapis nazywany, przy czym należy w tym pierwszym zapisie przestrzegać kolejności składowych i nie wolno zostawiać „wolnych miejsc". Wolno więc pisać:

  (22 , 6, Rok => 2000);

  (22 , Rok => 2000, Miesiac => 6);
- ⇝ W przypadku typów rekordowych istnieje możliwość nadawania wartości domyślnych niektórym składowym przez przypisanie tym składowym wyrażeń odpowiednich typów. Wyrażenia te muszą być tak skonstruowane, aby ich wartość była znana podczas kompilacji.

Weźmy pod uwagę następującą deklarację:

```
type Wspolrzedne_2D is
   record
      Pierwsza : Float := 0.0;
      Druga : Float := 0.0;
   end record;
```

Jeżeli zadeklarujemy zmienne:

```
Punkt : Wspolrzedne_2D;
Wersor_1 : Wspolrzedne_2D := (1.0, 0.0);
Wersor_2 : Wspolrzedne_2D := (Pierwsza => 0.0, Druga => 1.0);
```

to składowe Punkt.Pierwsza i Punkt.Druga mają wartości równe 0.0, a pozostałym zmiennym nadano wartości zgodnie z ich rolą w programie. Ponieważ w naszym przykładzie składowe są tego samego typu możemy stosować agregaty nazywane postaci:

```
(Pierwsza | Druga => 1.0)
(others => 1.0)
```

Liczba składowych rekordu nie jest ograniczona, a nawet dopuszczalne są rekordy puste tzn. z pustą listą pól. W takim przypadku musimy listę składowych zapisać jako null, albo napisać deklarację skróconą:

```
type Pusty_Rekord is null record;
```

Podanej deklaracji odpowiada agregat (null record).

W przypadku typów rekordowych ich składowymi nie mogą być stałe, ale możemy deklarować stałe typu rekordowego. Przykładem deklaracji stałych typu Wspolrzedne_2D mogą być następujace definicje:

```
Wersor_1 : constant Wspolrzedne_2D := (1.0, 0.0);
Wersor_2 : constant Wspolrzedne_2D := (0.0, 1.0);
```

Jak wspomnieliśmy, nie ma ograniczeń dotyczących typów składowych rekordu. W związku z tym możemy napisać:

```
with Ada.Text_Io; use Ada.Text_Io;
with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;

procedure Rekordy is
   type Data is
      record
         Rok : Positive range 1 .. 2500;
         Miesiac : Positive range 1 .. 12;
         Dzien : Positive range 1 .. 31;
      end record;
   type Osoba is
     record
       Data_Urodzin : Data;
       Pierwsze_Imie : String (1 .. 10) := (1 .. 10 => '␣');
       Drugie_Imie : String (1 .. 10) := (1 .. 10 => '␣');
       Nazwisko : String (1 .. 20) := (1 .. 20 => '␣');
    end record;
Studentka : Osoba;
begin
   Studentka.Data_Urodzin := (1981, 1, 1);
   Studentka.Pierwsze_Imie(1..4):= ("Anna");
   Studentka.Drugie_Imie(1..5):= ("Maria");
   Studentka.Nazwisko(1..6):= ("Zdolna");
   Put ("Dane␣studentki:"); New_Line;
   Put (Studentka.Pierwsze_Imie); Put('␣');
   Put (Studentka.Drugie_Imie); Put('␣');
   Put (Studentka.Nazwisko);
   Put (Studentka.Data_Urodzin.Rok, 4); Put ('.');
   Put (Studentka.Data_Urodzin.Miesiac, 2); Put ('.');
   Put (Studentka.Data_Urodzin.Dzien, 2);
end Rekordy;
```

Zwróćmy uwagę w jaki sposób odwołujemy się do danych opisujacych datę urodzenia oraz na sposób nadawania wartości danym dotyczących imion i nazwiska.

Zgodnie z naszym oczekiwaniem program wypisał napisy:

```
Dane studentki:
Anna Maria Zdolna 1981. 1. 1
```

Na tym kończymy omawianie typów strukturalnych, ale sygnalizujemy, że w dalszych rozdziałach pojęcia związane z tymi typami będą często stosowane w różnych kontekstach. W szczególności w rozdziale 8 pokazujemy, że typy rekordowe są niezbędnym narzędziem używanym do konstrukcji dynamicznych struktur danych, a w rozdziale 12 pokażemy w jaki sposób typy rekordowe wykorzystywane są do pisania programów zorientowanych obiektowo.

Uzupełnienie

Dla typów strukturalnych nie istnieją żadne operatory relacyjne poza operato-

rem równości (i oczywiście komplementarnym operatorem różności). Dwa rekordy lub tablice są sobie równe jeżeli wszystkie pola tych rekordów (wszystkie elementy tablic) są równe, w przeciwnym wypadku rekordy te (tablice) są różne.

Oczywiście pozostałe operatory relacyjne można sobie zdefiniować samodzielnie, w tym także zmienić definicję pojęcia równości. W celu ilustracji tej techniki posłużymy się przykładem sortowania.

Sortowanie polega na ustawieniu elementów pewnego zbioru skończonego w ciąg. Najczęściej elementami zbioru, który chcemy sortować są rekordy, w których jedną składową jest klucz elementu. Klucze mogą być porównywane ze sobą – są więc danymi typu, dla którego określona jest relacja częściowego porządku (np. $\leqslant$, a w Adzie $\Rightarrow$ <=). Relacja ta jest przenoszona na całe rekordy. Rekordy z danymi mogą być deklarowane jako dane następującego typu[3]:


```
type Element is
   record
      Klucz : Typ_Klucza;
      Dane : Typ_Danych;
   end record;
```

a tablica, którą mamy sortować może być np. następującego typu:

```
type Zbior is array (Integer range <>) of Element;
A : Zbior (1 .. N);
```

Mówimy, że tablica reprezentująca zbiór rekordów jest posortowana w porządku rosnącym jeżeli dla dowolnego indeksu J mamy

$$A_J.\mathrm{Klucz} \leqslant A_{J+1}.\mathrm{Klucz}, \quad J = 1, .., N-1.$$

Jeżeli relację $\leqslant$ zastąpimy przez $\geqslant$, to otrzymamy tablice posortowaną w porządku malejącym. Dalej będziemy dla ustalenia uwagi zajmować się wyłącznie sortowaniem w kierunku rosnącym.

Sortowaniem wewnętrznym nazywamy sortowanie tablicy małej na tyle, że wszystkie jej elementy mogą się zmieścić w pamięci operacyjnej. W przypadku *sortowania zewnętrznego* tablica sortowana nie mieści się w całości w pamięci operacyjnej i w związku z tym część rekordów przechowywana jest w pamięci zewnętrznej.

Przypuśćmy, że mamy dwa rekordy R1 i R2 umieszczone w A (I) i w A (J). Mówimy, że R1 poprzedza R2 jeżeli $I < J$. Jeżeli w nieposortowanej tablicy rekord R1 poprzedza R2 i R1.Klucz = R2.Klucz, to w tablicy posortowanej również R1 poprzedza R2. Innymi słowy sortowanie stabilnie zachowuje względne położenia rekordów o tych samych wartościach kluczy.

Sortowaniem w miejscu (*in situ*) nazywamy sortowanie, w którym tablica nieuporządkowania i uporządkowana zajmują ten sam obszar pamięci, przy czym do realizacji algorytmu mozna stosować pewien niewielki obszar pomocniczy. Inaczej mówiąc w przypadku sortowania *in situ* **nie** wykonuje się kopii sortowanej tablicy.

[3] W rozdziale 12 Czytelnik zapozna się z bardziej złożonym, ale pozwalającym na większe możliwości sposobem definiowania takich struktur.

Istnieje bardzo wiele algorytmów sortowania, z których jednym z najbardziej efektywnych jest algorytm *quicksort*, który jednak jest zbyt złożony jak na potrzeby tej książki i dlatego tu pokażemy przykład łatwego do zrozumienia algorytmu sortowania przez prosty wybór. Przypuśćmy, że mamy daną tablicę

```
A : Zbior (1 .. N);
```

która należy uporządkować rosnąco. Wybieramy z tej tablicy element o najmniejszej wartości klucza i wstawiamy go do A (A'First) (w naszym przypadku A (1)). Następnie przeglądamy tablicę od elementu drugiego i znów znajdujemy ten, który ma najmniejszy klucz. Wstawiamy ten element do A (2). Postępujemy tak aż do momentu, w którym cała tablica zostanie przetworzona.

Zaczynamy więc następująco: Bierzemy A (1).Klucz i porównujemy z A (2).Klucz. Jeżeli A (2).Klucz < A (1).Klucz, to wymieniamy A (1), z A (2) i porównujemy A (1).Klucz z A (3).Klucz, wymieniamy rekordy, jeżeli to konieczne, aż do momentu, kiedy porównaliśmy A (1).Klucz z A (N).Klucz i ewentualnie zamieniliśmy te rekordy. W tym miejscu mamy pewność, że w A (1) znajduje się element z najmniejszym kluczem.

Następnie powtarzamy ten cykl, z tym, że tym razem dla drugiego elementu tablicy.

Przy realizacji musimy często dokonywać wymiany rekordów. Dogodnie jest zatem użyć procedury Zamien, która może mieć formę przedstawioną poniżej w rozwiązaniu naszego problemu.

```
1  with Ada.Text_Io;
2  use Ada.Text_Io;
3  with Ada.Integer_Text_Io;
4  use Ada.Integer_Text_Io;
5  procedure Sortuj_Tablice_1 is
6
7     subtype Typ_Klucza is Character range 'A' .. 'Z'; -- Kluczem sa wielkie litery
8
9     type Element is
10       record
11          Klucz : Typ_Klucza;
12          -- Dane : Typ_Danych;
13       end record;
14
15    type Tablica is array (Integer range <>) of Element;
16
17    procedure Zamien (
18          T : in out Tablica;
19          K,
20          L : in Integer ) is
21       Temp : Element;
22    begin
23       Temp := T(K);
24       T(K) := T(L);
25       T(L) := Temp;
26    end Zamien;
27
28    procedure Czytaj_Tablice (
```

```
29          T : out Tablica ) is
30      Znak : Character;
31    begin
32      for Indeks in T'range loop
33        loop
34          Get (Znak);
35          exit when Znak in Typ_Klucza;
36          Put (" To nie byla wielka litera ");
37          New_Line;
38        end loop;
39        T(Indeks).Klucz := Znak;
40      end loop;
41    end Czytaj_Tablice;
42
43    procedure Wypisz_Tablice (
44          T : in Tablica ) is
45    begin
46      for Indeks in T'range loop
47        Put (T(Indeks).Klucz);
48      end loop;
49    end Wypisz_Tablice;
50
51    procedure Sortuj_Tablice (
52          T : in out Tablica;
53          I : out Natural ) is
54      -- Sortowanie przez proste wstawianie.
55      -- Jezeli tablica ma N elementow, to w najgorszym przypadku
56      -- musimy wykonac N*(N-1)/2 przestawien elementow.
57      -- Feldman, M. B. (1988). Data structures with Modula-2.
58      -- Prentice_Hall, London, 250, 251
59    begin
60      I := 0;
61      for K in T'First..T'Last-1 loop
62        for L in K+1..T'Last loop
63          if T(L).Klucz < T(K).Klucz then
64            Zamien (T, K, L);
65            I := I + 1;
66          end if;
67        end loop;
68      end loop;
69    end Sortuj_Tablice;
70
71    Max_Ilosc : constant := 10;
72    Ilosc_Elementow : Positive
73    range 1..Max_Ilosc;
74    Ilosc_Zamian : Natural;
75
76  begin
77    Put (" Podaj ilosc elementow tablicy ");
78    New_Line;
79    Get (Ilosc_Elementow);
80    declare
81      T : Tablica (1 .. Ilosc_Elementow);
82    begin
```

```
83          New_Line;
84          Put (" Wprowadz");
85          Put (Ilosc_Elementow,2);
86          Put (" wielkich liter");
87          New_Line;
88          Czytaj_Tablice (T);
89          New_Line;
90          Put (" Oto tablica przed sortowaniem : ");
91          New_Line;
92          Wypisz_Tablice (T);
93          Sortuj_Tablice (T, Ilosc_Zamian);
94          New_Line;
95          Put (" Wykonano ");
96          Put (Ilosc_Zamian, 3);
97          Put (" wymian ");
98          New_Line;
99          Put (" Oto tablica po sortowaniu : ");
100         New_Line;
101         Wypisz_Tablice (T);
102      end;
103   end Sortuj_Tablice_1;
```

Rozdział 6

Procedury i funkcje – podprogramy

Przy konstrukcji dużych i złożonych programów napotykamy na trzy podstawowe problemy:

1. wielkość programu,
2. poziom abstrakcji,
3. niekorzystne efekty uboczne dotyczące zmiennych powodowane przez akcje wykonywane w różnych częściach programu.

Rozwiązanie tych problemów umożliwiają języki programowania zapewniające:

- ⇝ dekompozycję programu na mniejsze, łatwiejsze do analizy i kontroli części,
- ⇝ możliwość deklarowania abstrakcyjnych typów danych,
- ⇝ kontrolę widzialności elementów programu.

Ogromną zaletą Ady jest między innymi to, że język ten zawiera mechanizmy umożliwiające realizację podanych wymagań.

Możliwość definiowania abstrakcyjnych typów danych została omówiona w poprzednich rozdziałach, natomiast spełnienie pozostałych dwóch wymagań umożliwiają pojęcia podprogramów i pakietów. W tym podrozdziale zajmiemy się *podprogramami*, które są bardzo ważną częścią każdego języka programowania, tak ważną, że bez niej programowanie jest prawie niemożliwe.

Rozważmy problem polegający na przetwarzaniu zbioru danych składających się z nagłówka i ciągu N podobnych, oddzielnych jednostek. Ogólnie można ten problem rozwiązać pisząc:

```
Czytaj_Naglowek;
Przetworz_Naglowek;
Pisz_Naglowek;
for I in 1 .. N loop
   Czytaj_Jednostke;
   Przetworz_Jednostke;
   Wypisz_Jednostke;
end loop;
```

Łatwo zauważyć, że rozwiązanie problemu jest opisane przy pomocy prostszych zadań, nazywanych czasami podzadaniami, dzięki czemu widoczna jest dobrze struktura algorytmu przetwarzania danych przy jednoczesnym ukryciu szczegółów. Jest jasne, że podzadania, takie jak np. Czytaj_Naglowek, Przetworz_Naglowek, muszą być opisane ze wszystkimi potrzebnymi szczegółami, ale zamiast zastępować nazwy tych podzadań długimi programami, można te nazwy traktować jako identyfikatory i określić szczegóły akcji wykonywanych w tych programach w oddzielnych częściach programu nazywanych podprogramami. Określenia te nazywane są deklaracjami podprogramów, gdyż określają akcję wykonywaną przez podprogram i nadają temu podprogramowi nazwę, czyli identyfikator podprogramu. Użycie identyfikatora podprogramu w operacyjnej części programu nazywane jest wywołaniem podprogramu i powoduje wykonanie akcji opisanych w deklaracji podprogramu.

Z podanego przykładu widać, że podprogramy umożliwiają jasny zapis struktury programu i dekompozycję programu na spójne logicznie części. Poza tym, podprogramy są szczególnie użyteczne w sytuacji, gdy ten sam algorytm realizowany przez podprogram jest wykorzystywany w wielu miejscach programu. Powoduje to zmniejszenie postaci źródłowej i wynikowej programu oraz redukuje możliwości popełnienia błędu.

6.1 Projektowanie podprogramów

Zauważmy, że treść procedury jest podobna do innych części programu, przy istotnej różnicy polegającej na tym, że treść procedury podana jest w części deklaracyjnej programu. Wydzielenie fragmentu programu i umieszczenie go w osobnym obszarze (w części deklaracyjnej) umożliwia ukrycie szczegółów implementacyjnych. Jeżeli wiemy jak wywołać podprogram i jaka jest jego funkcja nie musimy wiedzieć według jakiego algorytmu działa. Przykładem jest procedura Put, której używamy nie wiedząc, a nawet nie interesując się, jak napisano kod tej procedury.

Określenie tego co robi podprogram i jak należy go wywołać nazywamy *interfejsem podprogramu*. Drugim pojęciem związanym z projektowaniem podprogramów jest *ukrywanie informacji* o implementacji.

Projektowanie podprogramu powinno być podzielone na dwa zadania

1. projektowanie interfejsu
2. projektowanie implementacji.

W celu zaprojektowania interfejsu należy **ściśle** określić funkcje podprogramu i sposób komunikacji z podprogramem. W celu zaprojektowania implementacji musimy określić algorytm według którego podprogram wykonuje swoje funkcje. Przy projektowaniu interfejsu dobrze jest sporządzić listę następujących elementów

1. Wartości, które procedura otrzymuje z podprogramu wywołującego.
2. Wartości, które procedura oblicza i przekazuje do podprogramu wywołującego
3. Wartości, które procedura otrzymuje z programu wywołującego, zmienia je i przesyła po zmianie do podprogramu wywołującego

Jeżeli mamy taką listę, to możemy utworzyć listę parametrów formalnych, przy czym parametry pierwszej grupy są rodzaju wejściwego, drugiej – wyjściowego, trzeciej – wejściowo/wyjściowego.

6.2 Zagnieżdżanie

Obszar deklaracji jest deklaracją podprogramu razem z jego treścią. Identyfikatory zadeklarowane wewnątrz obszaru deklaracji nazywany *lokalnymi w tym obszarze*. Jeżeli procedury są zagnieżdżone, to dostaniemy zagnieżdżone obszary deklaracji.

Deklaracje występujące w najbardziej zewnętrznym obszarze deklaracji nazywamy *globalnymi* w stosunku do wewnętrznych obszarów deklaracji. Można używać tych samych identyfikatorów w deklaracjach pod warunkiem, że identyfikatory występują w *różnych obszarach deklaracji*. Identyfikatory takie nazywamy czasami *homografami*. Mimo, że nazwy są takie same, to związane są z innymi obszarami pamięci. Dostęp do poszczególnych homografów ilustruje poniższy program:

```
1  with Text_IO; use Text_IO;
2
3  procedure Homograf is
4     package IIO is new Integer_IO (Integer);
5     use IIO;
6
7     Zm : Integer := 1;
8
9     procedure Lokalna;
10    procedure Lokalna is
11       Zm : Integer := 2;
12
13       procedure Lokalna_2;
14       procedure Lokalna_2 is
15          Zm : Integer := 3;
16       begin
17          Put (Zm); -- Wypisuje 3
18          New_Line;
19          Put (Lokalna.Zm); -- Wypisuje 2
```

```
20            New_Line;
21            Put (Homograf.Zm); -- Wypisuje 1
22            New_Line;
23            Put (Homograf.Lokalna.Zm); -- Wypisuje 2
24            New_Line;
25         end Lokalna_2;
26
27      begin
28         Lokalna_2;
29      end Lokalna;
30
31   begin
32      Lokalna;
33   end Homograf;
```

6.3 Reguły zasięgu

Reguły zgodnie z którymi można używać identyfikatora nazywamy *regułami zasięgu* identyfikatorów.

1. *Zasięgiem identyfikatora* nazywamy wszystkie instrukcje występujące po definicji identyfikatora i znajdujące się w obszarze deklaracji z wyłączeniem punktu 2.
2. Zasięg identyfikatora nie obejmuje zagnieżdżonego obszaru deklaracji, w którym umieszczono deklarację odpowiedniego homografu.

Ta druga reguła jest nazywana czasami *priorytetem identyfikatorów*

6.4 Kilka definicji

Parametry podprogramów służą do przesyłania informacji do i z procedury. Dla procedury zdefiniowanej jako

procedure Get (Item : out Num; Width : in Field := 0)

Następujące wywołanie: Get (Item => Liczba) powoduje przesłanie wartości z procedury do zmiennej Liczba, a wcześniej z procedury wywołującej do procedury Get parametru Width o wartości równej zero.

Podobnie jak zmienne parametry są nazwami obszarów pamięci używanych do przechowywania wartości danych. Item jest nazwą innego obszaru niż obszar przypisany do Liczba. Po zakończeniu procedury Get wartość z Item jest kopiowana do Liczba[1].

Parametrami formalnymi nazywamy zmienne zadeklarowane w nagłówku procedury. Określenie parametru formalnego musi zawierać identyfikator i typ danych.

W deklaracji parametrów formalnych nie można używać zakresów.

[1] Możliwa jest także taka implementacja, w której obszary te są tożsame.

Parametrami aktualnymi nazywamy zmienne albo wyrażenia związane z nazwą parametru formalnego przy wywoływaniu procedury. Typ parametru aktualnego **musi odpowiadać** parametrowi formalnemu.

Związaniem nazywanym nazywamy związanie parametrów aktualnych z formalnymi, poprzez jawne podanie nazw parametrów formalnych przy wywołaniu podprogramu. W takim przypadku porządek parametrów przy wywoływaniu nie musi być taki sam jak na liście parametrów formalnych w nagłówku procedury. Jeżeli jednak zapomnimy o jakimś z nich – kompilator się o niego upomni.

Związaniem pozycyjnym nazywamy związanie parametrów aktualnych z formalnymi przez ich pozycje na liście parametrów aktualnych i formalnych. W tym wypadku na liście parametrów aktualnych znajdują się tylko ich desygnatory, przy czym ich kolejność musi być taka sama jak odpowiednich parametrów formalnych na liście w nagłówku podporogramu. Zwiazanie pozycyjne jest w wielu wypadkach gorsze od związania przez nazwę. Przy stosowaniu związania pozycyjnego łatwo pomylić porządek parametrów, a błędy z tego wynikające mogą być trudne do wykrycia. Poza tym związanie przez nazwy daje bardziej czytelną formę programu. Istnieją jednak przypadki, kiedy dodatkowa informacja zawarta w związaniu nazywanym nie jest specjalnie przydatna. Dotyczy to szczególnie podprogramów z jednym parametrem, albo takich procedur jak Zamien, kiedy sama nazwa i ilość parametrów mówią nam co zostało wykonane podczas wywołania podprogramu.

Interfejsem nazywamy formalną definicję celu działania podprogramu i mechanizmu komunikowania się z podprogramem.

Ukrywaniem informacji nazywamy ukrywanie szczegółów implementacyjnych modułu programowego.

W Adzie mamy dwa rodzaje podprogramów: *procedury* i *funkcje* (także ich specjalne wersje – *operatory*).

6.5 Efekty uboczne

Wpływ jednego modułu na inny, poza jawnie określonym interfejsem tych modułów nazywamy *efektem ubocznym*. Przez moduł programowy rozumiemy tu procedurę, funkcję, albo podprogram główny.

W dotychczasowych rozważaniach przyjęliśmy, że funkcja oblicza pewną wartość na podstawie dostarczonych danych. I tak np. funkcja sin (x) oblicza wartość sinusa pewnego kąta, a wynik jej działania zależy tylko i wyłącznie od wartości parametrów. Dlatego też w języku Ada, w przeciwieństwie do innych języków, parametry funkcji muszą być przekazywane w trybie in[2]. Rzecz jasna życie nie

[2]Funkcja może zmieniać wartości parametrów jeżeli są one przekazywane poprzez wskaźniki (rozdział 8). W takim wypadku nie jest zmieniany parametr funkcji, ale może być zmieniana wartość wskazywana przez ten wskaźnik. Efekt takiego zabiegu jest tożsamy z tym, który uzyskuje się w innych językach programowania poprzez przekazywanie parametrów poprzez referencję (w

jest takie proste i bardzo często zdarza się, że wynik obliczenia funkcji jest zależny nie tylko od aktualnych parametrów funkcji, ale także od wyników poprzednich obliczeń. W takim wypadku może się okazać, że oprócz obliczania wyniku funkcja zmienia także wartość pewnej zmiennej w bloku otaczającym naszą funkcję (np. zmienną globalną). Takie działanie nazywane jest *efektem ubocznym* działania funkcji (także procedury). Np. jeżeli kolejne wywołania bezparametrowej funkcji Silnia mają obliczać kolejne wartości tej funkcji to może ona mieć postać:

```
1 Iloczyn : Integer := 1; Ostatnia_Wartość := 0; ... function Silnia
2 return Integer is begin
3     Ostatnia_Wartosc := Ostatnia_Wartosc + 1;
4     Iloczyn := Iloczyn * Ostatnia_Wartosc;
5     return Iloczyn;
6 end Silnia;
```

Jak widać kolejne wywołania funkcji Silnia będą obliczać różne wyniki.

Wykorzystywanie efektów ubocznych procedur (**ale nie funkcji**) jest bardzo wygodnym i dość bezpiecznym mechanizmem, stosowanym bardzo często, ponieważ umożliwia zmniejszenie liczby parametrów przekazywanych do danej procedury, a trzeba pamiętać, że duża liczba parametrów zmniejsza znakomicie czytelność programu [3]. Z drugiej jednak strony, stosowanie efektów ubocznych w programach wielozadaniowych (rozdział 14), w stosunku do zmiennych globalnych grozi popełnieniem, bardzo trudnych do znalezienia, rzadko się objawiających[4] błędów. Natomiast technika taka jest zawsze bezpieczna, jeżeli dotyczy zmiennych lokalnych procedury otaczającej[5] (rozdział 6.9).

Inaczej wygląda sprawa w przypadku funkcji, gdzie ten sam mechanizm może powodować bardzo kłopotliwe i czasami zaskakujące zachowanie programu.

Wyjaśnienie tego pokażemy na prostym przykładzie:

```
1 y : Float; ... function f (x : Float) return Float is begin
2     ...
3     y := 1.0;
4     return x*y;
5 end f; ... function g (x : Float) return Float is begin
6     ...
7     y := x*y;
8     return y;
9 end g;
```

Rodzi się pytanie. Jaką wartość obliczy wyrażenie: f (2.0)* g (3.0) i jaką wartość będzie miała zmienna y?

Zapewne większość czytelników odpowie, że będzie to wartość 6.0. Niestety może się okazać, że nie zawsze tak będzie! Dlaczego?! A dlatego, że kolejność

Adzie – tryb in out). Stanowczo jednak odradzamy stosowanie takich technik, choć zdajemy sobie sprawę, że są one konieczne w niektórych sytuacjach – np. przy współpracy z procedurami systemu operacyjnego (np. Windows API).

[3] Dobry obyczaj każe przy nagłówku każdej procedury stosującej efekty uboczne – opisać je w taki sposób, żeby ktoś, kto czyta program zdawał sobie z nich sprawę i w ten sposób ułatwić mu zrozumienie jego działania

[4] ale zgodnie z prawem Murphy'ego — zawsze w krytycznych momentach...

[5] Pamiętajmy, że należy w miarę możliwości minimalizować liczbę zmiennych globalnych w programie. Innymi słowy: **Stosujmy zmienne lokalne!**

obliczania podwyrażeń w danym wyrażeniu **nie jest określona**. Zatem nie wiadomo, czy najpierw zostanie obliczona wartość funkcji f (2.0), czy g (3.0). W tym drugim wypadku, ze względu na to, że zmienna y nie jest zainicjowana wartość naszego wyrażenia jest zupełnie przypadkowa.

Czytelnik zapewne z niedowierzaniem odnosi się do przedstawionego poglądu stwierdzając, że nie spotkał się jeszcze z przypadkiem takiego „złośliwego" zachowania się kompilatora. A jednak bardzo często kod generowany przez kompilator przy włączonej opcji generowania informacji dla programu uruchomieniowego (tzw. *odpluskwiacza*, czyli *debuggera*) jest wygenerowany inaczej niż kod wygenerowany na podstawie tego samego programu, przez ten sam kompilator przy wyłączonej tej opcji, a włączonej optymalizacji kodu. Jeżeli program jest generowany przez inny kompilator, to już nic nie można domniemać na temat kolejności obliczania podwyrażeń[6].

Typowym przykładem jest zmiana wartości zmiennej globalnej w podprogramie. Aby uniknąć trudnych do wykrycia błędów wywołanych przez efekty uboczne przesyłanie informacji pomiędzy podprogramami powinno odbywać się **wyłącznie poprzez interfejs**.

Deklarując zmienne po treściach procedur zapewniamy brak dostępu do tych zmiennych w procedurach.

Podprogram nie korzystający ze zmiennych globalnych (lub otaczających dany podprogram) z całą pewnością **nie** generują efektów ubocznych. Takie programy są czasem nazywane *reentrant*. Szczególną uwagę w stosowaniu efektów ubocznych należy poświęcić w programach, w których działa wiele współbieżnie działających zadań (rozdział 14).

6.6 Funkcje

Omawiając wyrażenia wspomnieliśmy, że funkcje są podprogramami, które przy wywołaniu są argumentami tych wyrażeń. Przykładami wywołań funkcji są więc Sqrt(2.0), Sin(Kat), Cos(Kat), które używaliśmy w niektórych przykładach. W tym podrozdziale zajmiemy się przede wszystkim deklarowaniem funkcji, oraz ich wykorzystaniem w programach, przy czym należy podkreślić, że w pakietach bibliotecznych Ady znajdziemy bardzo wiele funkcji, z których należy korzystać zamiast pisać własne funkcje, które nie zawsze będą równie efektywne jak ich odpowiedniki biblioteczne. Typowa deklaracja funkcji ma postać:

```
function Nazwa (Parametr_1 : Typ_Parametru_1;
                Parametr_2 : Typ_Parametru_2;
                ...
                Parametr_N : Typ_Parametru_N)
                      return Typ_Wyniku is
   -- Deklaracje lokalne
begin
   -- Instrukcje
```

[6]Nawet kolejne wersje kompilatorów tego samego producenta potrafią wygenerować inaczej działający program na podstawie tego samego kodu źródłowego, czego znakomitym przykładem są najbardziej popularne kompilatory Microsoft Visual C++, czy Borland C++. Chwalebnym wyjątkiem są kompilatory GNU (w tym także kompilator Ada GNAT), a także kompilatory języka Java

```
   -- Instrukcje
   return Wynik;
   -- ewentualna sekcja obsługi wyjątków
end Nazwa;
```

Deklarację funkcji zaczynamy słowem kluczowym function, po którym podajemy identyfikator funkcji, czyli jej nazwę i tę nazwę stosujemy wywołując funkcję w naszym programie. Po identyfikatorze funkcji podano w nawiasach okrągłych listę parametrów formalnych, przy czym funkcja może nie zawierać tej listy. Następnie mamy słowo kluczowe return, po którym określamy typ wyniku funkcji, a więc typ wartości wyznaczanych przez funkcję. Należy tu dodać, że Typ_Wyniku nie może zawierać jawnego ograniczenia.

Część deklaracji zaczynającą się od słowa function i kończącą się przed słowem is nazywamy *nagłówkiem*, albo *opisem funkcji* (ang. specification). Informacje zawarte w nagłówku wystarczają do prawidłowego wywoływania funkcji w wyrażeniach. Słowo is rozpoczyna *treść funkcji*[7]. Treść składa sie z *części deklaracyjnej* i *części operacyjnej*, którą jest ciąg instrukcji ujęty w nawiasy syntaktyczne begin .. end. Część deklaracyjna zawiera wszystkie deklaracje, w tym deklaracje podprogramów, potrzebne oprócz parametrów formalnych do zapisania algorytmu realizowanego przez funkcję. Część deklaracyjna funkcji nie jest konieczna, natomiast ciąg instrukcji musi zawierać co najmniej jedną instrukcję, przy czym przynajmniej jedna instrukcja musi być instrukcją postaci return Wyrazenie. Wyrażenie to musi być tego samego typu co Typ_Wyniku podany w nagłówku. Należy dodać, że możemy mieć kilka instrukcji return, nazywanych też instrukcjami powrotu i wykonanie dowolnej z nich powoduje zakończenie akcji wykonywanych przez funkcję. Jako dobry przykład może służyć funkcja Signum, która ilustrowała zastosowanie instrukcji if.

Każde wywołanie funkcji powoduje kreację obiektów deklarowanych lokalnie przez funkcję oraz parametrów i te obiekty nikną w momencie, gdy funkcja się kończy. Dzięki temu funkcja może wywoływać samą siebie i w ten sposób dochodzimy do pojęcia rekurencji, któremu poświęcamy następny rozdział.

Wróćmy na chwilę do listy parametrów formalnych. Parametr formalny może być dowolnego typu, ale typ ten musi mieć nazwę. Jedynym wyjątkiem od tej reguły są parametry typów wskaźnikowych, które omawiane są w rozdziale 8. Typ parametru formalnego może być typem tablicowym otwartym. Poprawne są więc deklaracje:

```
type Wektor is array (Integer range <>) of Float;

function Iloczyn_Skalarny (A, B : Wektor) return Float is
   Wynik : Float := 0.0;
begin
   if A'First /= B'First or A'Last /= B'Last then
      raise Constraint_Error;
   end if;
   for I in A'Range loop
      Wynik := Wynik + A(I)*B(I);
   end loop;
   return Wynik;
```

[7] ang. *body* znaczy tu „treść”, a nie „ciało”!

```
end Iloczyn_Skalarny;
```

Zauważmy, że w funkcji Iloczyn_Skalarny występuje instrukcja warunkowa służąca do sprawdzenia, czy wymiary mnożonych wektorów są takie same i w przypadku, gdy tak nie jest, następuje zgłoszenie wyjątku CONSTRAINT_ERROR. Można zadeklarować własny wyjątek np. Niezgodne_Wymiary i odpowiednią procedurę jego obsługi. Te zagadnienia omawiane są w drugiej części naszego kursu programowania w Adzie.

W odróżnieniu od języka Pascal, typ wyniku obliczanego przez funkcje Ady może być typem strukturalnym np. tablicą otwartą. Przykładem takiego podprogramu jest funkcja, która tworzy z wektora wejściowego (parametru funkcji) wektor (wartość funkcji), którego składowe występują w odwróconym porządku do porządku składowych wektora wejściowego.

```
1 function Odwroc_Wektor (V : Wektor) return Wektor is
2   Wynik : Wektor(V'Range);
3 begin
4   for I in V'Range loop
5     Wynik(I) := Wynik(V'First + V'Last−I);
6   end loop;
7   return Wynik;
8 end Odwroc_Wektor;
```


6.7 Operatory

Specjalnymi wersjami funkcji są operatory, czyli funkcje wywoływane niejawnie definiujące operacje m.in. dodawania, odejmowania niezdefiniowane dla pewnych typów danych. Rzecz jasna wiadomo, co to znaczy $a + b$ jeżeli a, b są typu całkowitoliczbowego albo zmiennoprzecinkowego. Jeżeli jednak dane te są macierzami albo wektorami to kompilator «nie wie» co rozumieć przez operację dodawania i operację tę należy zdefiniować. Podobnie jeżeli zarówno a jak i b są różnych typów np. a jest typu określającego dokładną datę (będzie to prawdopodobnie rekord) zaś b będzie np. liczbą całkowitą określającą liczbę dni.

Można zdefiniować wszystkie operatory występujące w języku Ada[8] (tj. and, or, xor, =, /=, <, >, <=, >=, +, − [9], &, *, /, mod, rem, **, abs, not) w następujący sposób:

```
function "*" (Lewy : Wektor; Prawy : Float) return Wektor is
  Wynik : Wektor( Lewy'range );
begin
  for i in Wynik'range loop
    Wynik(i) := Lewy(i)*Prawy;
  end loop;
  return Wynik;
end "*";
```

[8] ale nie and then, ani or else, ani też new.

[9] Warto pamiętać, że operatory „+” i „−” są zarówno operatorami binarnymi (np. a+b) jak i unarnymi (np. −a)

O tej chwili można używać konstrukcji:

```
a := b*3.0;
```

gdzie a i b są odpowiednio zdefiniowanymi wektorami.

Czytelnik zapewne odgadnie dlaczego nie można użyć konstrukcji

```
a := 3.0*b;
```

ani

```
a := b*3;
```

Trzeba wspomnieć, że w przeciwieństwie do innych języków zdefiniowanie operatora „=" definiuje jednocześnie operator „/=" i odwrotnie – zdefiniowanie operatora „/=" definiuje jednocześnie operator „=".

Uzupełnienie

Bardzo pożytecznym, a niedocenianym operatorem jest unarny +. Oczywiście wiadomo, że jeżeli zmienna a jest typu całkowitego (rozdział 3.7) lub rzeczywistego (rozdział 3.8), to +a jest zawsze równe a. Ale jeżeli a jest np. napisem, rekordem lub jakąkolwiek inną zmienną operator ten możemy wykorzystać do swoich celów

Np. jeżeli zdefiniujemy następujący pakiet (rozdział 9):

```
1  package Uni is
2
3     ----------------------------------------
4     -- Funkcja zamienia napis na napis w kodzie
5     -- UNICODE i dodaje na końcu znak o kodzie
6     -- ASCII.nul (0), który jest znakiem
7     -- kończącym napis w wielu systemach operacyjnych
8     function "+" (Arg : String) return Wide_String;
9
10    ----------------------------------------
11    -- Funkcja wykonująca operację odwrotną...
12    function "+" (Arg : Wide_String) return String;
13
14 end Uni;
```

którego (bardzo uproszczona) implementacja będzie następująca:

```
1  with Ada.Characters.Handling; use Ada.Characters.Handling;
2
3  package body Uni is
4
5     function "+" (Arg : String) return Wide_String is
6     begin
7        return To_Wide_String (Arg & ASCII.nul);
8     end "+";
9
10    function "+" (Arg : Wide_String) return String is
11       s : String := To_String (Arg);
12    begin
13       if s (s'Last) = ASCII.nul then
```

```
14              return s (s'First .. s'Last - 1);
15          else
16              return s;
17          end if;
18      end "+";
19
20  end Uni;
```

to użycie takiego operatora może być następujące:

```
...
declare
    s : String := "Napis";
    w : Wide_String := +s;
begin
    ...
```

gdzie operator + pełni funkcję operatora konwersji. Jest to bardzo powszechne zastosowanie tego operatora.

6.8 Procedury

Procedury są drugim rodzajem podprogramów w Adzie. Przypominamy, że wywołanie procedury jest instrukcją. We wstępie do niniejszego rozdziału mielismy przykład, w którym pojawiły się wywołania procedur takie jak Czytaj_Naglowek, Przetworz_Naglowek. Poza tym, w poprzednich rozdziałach, w programach ilustrujących omawiane zagadnienia stosowaliśmy wywołania różnych procedur do czytania i wypisywania wyników obliczeń. Przykładami są instrukcje:

```
Put ("Podaj liczbe calkowita");
New_Line;
Get (A);
New_Line(2);
Put ("Liczba = ");
Put (A, 2);
```

Deklarowanie procedur bardzo przypomina deklarowanie funkcji. Deklarację procedury zaczynamy słowem kluczowym procedure, po którym podaje się identyfikator procedury i ewentualnie jej parametry formalne. Te trzy elementy tworzą *nagłówek procedury*. Po nagłówku możemy, w zależności od naszych potrzeb, podać deklaracje lokalne, a następnie po słowie begin jest część operacyjna procedury w formie ciągu instrukcji opisujących algorytm realizowany przez procedurę. Deklaracja procedury kończy się słowem end i identyfikatorem procedury.

Jak wspomniano, w deklaracji procedury mogą występować parametry formalne. W Adzie parametry formalne procedur mogą być trzech rodzajów: wejściowe (in), wejściowo-wyjściowe (in out) i wyjściowe (out). Rodzaj deklarowanego parametru formalnego określamy pisząc po dwukropku występującym po nazwie parametru odpowiednie słowa kluczowe: in, in out, albo out. Oczywiście parametr aktualny parametru przesyłanego w trybie wyjściowym, albo

wejściowo-wyjściowym **nie może** być wyrażeniem i **musi** być zmienną. Zadeklarowanie parametrów formalnych umożliwia wywoływanie procedury w różnych miejscach programu dla różnych wartości zmiennych określonych w miejscu wywołania. Jeżeli słowo wyróżniające rodzaj parametru nie występuje, to kompilator przyjmuje, że parametr jest parametrem wejściowym. W przypadku funkcji parametry formalne zawsze są parametrami wejściowymi i dlatego w naszych przykładach nie pisaliśmy słowa in. Mogliśmy jednak te nagłówki zapisać następująco:

```
function Iloczyn_Skalarny (A, B : in Wektor) return Float
function Odwroc_Wektor (V : in Wektor) return Wektor
```

Parametry formalne procedur mają następujące własności:

- ⇝ parametr wejściowy jest stałą inicjowaną przez wartość odpowiedniego parametru aktualnego i stąd wynika, że zmiana wartości parametru wejściowego procedury (albo funkcji) nie jest możliwa w części operacyjnej procedury (funkcji)[10]
- ⇝ parametr wejściowo-wyjściowy jest zmienną inicjowaną przez parametr aktualny i możliwe jest pobieranie oraz nadawanie wartości odpowiedniemu parametrowi aktualnemu w części operacyjnej procedury,
- ⇝ parametr wyjściowy jest niezainicjowaną zmienną i umożliwia nadanie wartości odpowiedniemu parametrowi aktualnemu w części operacyjnej procedury.

Stąd wynika, że parametry wejściowo-wyjściowe i wyjściowe należy traktować jak zwykłe zmienne lokalne podprogramu z uwzględnieniem podanych różnic dotyczących ich inicjacji.

Po tej dyskusji różnic pomiędzy funkcjami i procedurami dochodzimy do wniosku, że procedura jest pojęciem ogólniejszym od funkcji i funkcje mogą być łatwo przekształcone na odpowiednie procedury. Dla przykładu weźmy pod uwagę następującą procedurę:

```
1  procedure Iloczyn_Skalarny (A, B : in Wektor; S: out Float) is
2  begin
3     S := 0.0;
4     if (A'First /= B'First) or (A'Last /= B'Last) then
5        raise Constraint_Error;
6     end if;
7     for I in A'Range loop
8        S := S + A(I)*B(I);
9     end loop;
10 end Iloczyn_Skalarny;
```

W tym przypadku formalny parametr wyjściowy S służy do przekazania wyniku na zewnątrz procedury, ale podstawowa różnica polega na sposobie użycia procedury i funkcji. Wywołanie funkcji musi wystąpić w wyrażeniu np.:

[10] Inaczej niż w większości innych języków programowania. Rozwiązanie takie zostało podyktowane względami efektywnościowymi — kompilator sam decyduje, czy zmienną przekazuje jako kopię parametru aktualnego umieszczoną na stosie (tak jak to jest w C, Moduli-2, czy w Pascalu), czy przekazuje wyłącznie wskaźnik do zmiennej

```
Iloczyn := Iloczyn_Skalarny (Wektor_1, Wektor_2);
```

natomiast wywołanie procedury jest instrukcją, która w rozważanym przykładzie przyjmuje postać:

```
Iloczyn_Skalarny (Wektor_1, Wektor_2, Iloczyn);
```

Nie oznacza to jednak, że zalecamy stosowanie procedur w miejsce funkcji. Wręcz przeciwnie, uważamy, że funkcje w Adzie są bardzo użyteczne i należy je stosować wszędzie tam gdzie jest to naturalne. Warto tu dodać, że mechanizm podziału parametrów formalnych procedur na trzy wymienione rodzaje i ograniczenie parametrów formalnych funkcji jedynie do klasy parametrów wejściowych zabezpiecza w znacznym stopniu przed generowaniem tzw. efektów ubocznych przez podprogramy, a szczególnie przez funkcje[11]. Kolejny raz spotykamy własność, która świadczy o wyższości Ady w stosunku do Pascala, czy nawet Moduli-2.

W przykładach wywołań procedur i funkcji lista parametrów aktualnych była co do porzadku zgodna z listą parametrów formalnych. Możemy w miejsce zapisu pozycyjnego stosować zapis podobny do agregatów nazywanych i wtedy kolejność parametrów aktualnych może być różna od kolejności parametrów formalnych. Dopuszczalne jest stosowanie notacji mieszanej, ale wtedy pierwsza część listy musi być w notacji pozycyjnej i nie wolno stosować części others. Wracając do naszych przykładów moglibyśmy napisać:

```
Iloczyn := Iloczyn_Skalarny (A => Wektor_1, B => Wektor_2);,
Iloczyn_Skalarny (Wektor_1, B => Wektor_2, S => Iloczyn);
```

6.9 Zasięg i widzialność

Ważnym pojęciem związanym z procedurami jest *lokalność identyfikatorów*. Pojęcie to uwidacznia znaczenie procedur przy strukturalizacji programu i jego podziału na oddzielne, spójne logicznie części.

Jeżeli przeanalizujemy tekst funkcji Iloczyn_Skalarny, to zauważymy natychmiast, że zmienna Wynik jest istotna wyłącznie w treści tej procedury. Lokalność znaczenia tej zmiennej powinna być wyraźnie uwidoczniona i można to osiągnąć deklarując tę zmienną w części deklaracyjnej funkcji, tak jak zrobiliśmy. Zmienną Wynik nazywamy w takiej sytuacji zmienną lokalną. Jak Czytelnik już zapewne zauważył, deklaracja procedury przypomina oddzielny program. Deklaracje dopuszczalne w programie takie jak deklaracje stałych, typów, zmiennych i procedur mogą również występować w deklaracji procedury. Wynika stąd, że deklaracje procedur mogą się zagnieżdżać i mogą być określane rekurencyjnie.

Dobrym nawykiem programistycznym jest deklarowanie obiektów lokalnych, czyli ograniczenie ich istnienia do podprogramu, w którym są istotne. Podprogram, w którym zadeklarowano nazwę obiektu nazywamy *zasięgiem* tego obiektu. Ponieważ, jak wspomniano przy omawianiu instrukcji declare, deklaracje

[11] W przeciwieństwie do innych języków programowania Ada zniechęca jak może do korzystania z tzw. „efektów ubocznych".

mogą być zagnieżdżone, więc zasięgi mogą być również zagnieżdżone. Lokalność identyfikatorów umożliwia użycie tego samego identyfikatora dla różnych obiektów pod warunkiem, że nie doprowadzi to do niejednoznaczności[12].

Weźmy pod uwagę następujący fragment programu:

```
procedure A is
  X : Float := 0.0;
  ..
  procedure B is
    X : Float := 1.0;
    Y : Float := A.X;
  begin
    ..
  end B;
  ..
  procedure C is
    procedure D is
    begin
    ..
    end D;
  begin
    ..
  end C;
begin
..
end A;
```


Zauważmy, że procedura A posiada dwie lokalne procedury B i C, a C posiada z kolei procedurę lokalną D. Stąd wynika, że wywołanie procedury D w procedurze A jest niedopuszczalne[13], natomiast można wywołać A w D. Procedura D może być wywołana tylko w C i w samej procedurze D (rekurencja). Weźmy teraz pod uwagę deklarację zmiennych oznaczonych identyfikatorem X występujące w procedurach A i B. Mówimy, że wewnętrzna zmienna X zadeklarowana w B przesłania zewnętrzną zmienną X zadeklarowaną w A. W takiej sytuacji zewnętrzna zmienna X nie jest *bezpośrednio widzialna* w B. Można jednak odwołać się do tej zmiennej stosując zapis A.X (zmienna Y przyjmuje więc wartość 0.0, a nie 1.0) i wtedy zewnętrzne X jest nazywane *zmienną widzialną* w B.

Nieco inny problem powstaje, gdy w tym samym poziomie deklarujemy dwie procedury, powiedzmy A i B, które w swoich częściach operacyjnych zawierają wzajemne wywołania. Ponieważ deklaracja procedury musi zawierać zarówno nagłówek jak i część operacyjną oraz obowiązuje zasada liniowego porządku interpretowania deklaracji, instrukcja wywołania procedury B A przed deklaracją B jest niedopuszczalna. Problem daje się jednak rozwiązać w następujący sposób[14]:

```
procedure A (...); -- nagłówek A
```

[12] Sygnalizowane już pojęcie przeciążania, a w szczególności przeciążanie podprogramów omawiamy w następnym podrozdziale.

[13] po prostu jej nie widać!

[14] Dobry obyczaj nakazuje wymienienie wszystkich nagłówków procedur na początku programu i skomentowanie ich funkcji.

```
procedure B (...) is
begin
  ..
  A (...); -- wywołanie A
  ..
end B;

procedure A (...) is -- cała deklaracja A
begin
  ..
  B (...); -- wywołanie B
  ..
end A;
```

Zwróćmy uwagę na to, że procedura A została zapowiedziana przez podanie jej nagłówka, który, jak już wspomnieliśmy, zawiera wszystkie informacje konieczne do jej poprawnego użycia, następnie podano deklarację B i w końcu zadeklarowano procedurę A, łącznie z powtórzonym nagłówkiem, który musi być identyczny jak w zapowiedzi.

Lokalność identyfikatorów jest ważnym i użytecznym mechanizmem programowania, gdyż umożliwia konstruowanie złożnych programów w zespole kilku programistów, którzy mogą pisać swoje procedury niezależnie. Informacjami które muszą wymieniać są nagłówki procedur, a nie całe ich teksty.

Reasumując, lokalność i reguła, że zmienna nie istnieje poza swoim zasięgiem, implikują, że wartość zmiennej jest tracona, gdy procedura (funkcja), w której została zadeklarowana zmienna jest zakończona. To powoduje, że przy ponownym wywołaniu tej procedury wartość zmiennej jest nieznana i musi być ponownie utworzona. Jeżeli więc chcemy, aby wartość zmiennej została zachowana pomiędzy dwoma wywołaniami procedury, to zmienna ta musi być zadeklarowana poza procedurą. W związku z tym, *czasem życia zmiennej* nazywamy *czas aktywności procedury*, w której zmienna jest zadeklarowana.

Używanie lokalnych deklaracji ma trzy podstawowe zalety:

1. Uwidacznia fakt, że znaczenie obiektu jest ograniczone do procedury, która jest zazwyczaj niewielką częścią programu.
2. Zapewnia, że mimowolne użycie obiektu lokalnego poza jego zasięgiem jest wykrywane przez kompilator.
3. Zapewnia optymalizację gospodarowania pamięcią. Pamięć rezerwowana na obiekty lokalne jest zwalniana po wykonaniu procedury określającej ich zasięg i może być wykorzystana następnie dla innych zmiennych.

Na zakończenie tego podrozdziału podajemy sugestie dotyczące nazywania procedur i funkcji. Identyfikatorami funkcji powinny być rzeczowniki, które oznaczają ich wynik w wyrażeniach zawierających wywołania tych funkcji. Identyfikatorami funkcji o wartościach logicznych powinny być przymiotniki. Zwykłe procedury powinny mieć czasowniki jako identyfikatory opisujące akcje wykonywane przez te procedury.

6.10 Przeciążanie podprogramów

W matematyce stosujemy te same operatory do oznaczania formalnie różnych działań. Dla przykładu, działanie $a + b$ może oznaczać dodawanie w ciele liczb rzeczywistych, albo dodawanie elementów pewnej przestrzeni wektorowej. Za każdym razem, gdy pojawia się tego typu zapis, wiemy z dodatkowych informacji dostarczanych przez autora, co dokładnie ten zapis oznacza. Podobna sytuacja występuje przy tworzeniu wyrażeń w Adzie. Pamiętamy, że ten sam operator + stosowany jest przy dodawaniu argumentów typu Float oraz typu Integer. Mówimy w takim przypadku o *przeciążaniu operatora* dodawania, lub innych operatorów. Powstaje natychmiast pytanie czy w Adzie istnieje taki mechanizm przeciążania operatorów, który dopuszcza zapis $a + b$ w sytuacji, gdy obydwa argumenty są tego samego typu strukturalnego np. tablicami jednowymiarowymi tej samej długości. Odpowiedź jest twierdząca, a mechanizmem tym jest przeciążanie funkcji. Dokładniej, w Adzie możemy deklarować funkcje, których „identyfikatorami" są odpowiednie nazwy operatorów ujęte w cudzysłowy np. "+". Następujące operatory mogą być stosowane do nazywania takich funkcji:

6

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

Jeżeli ktoś uzna, że znak „*" powinien w jego programie, czy pakiecie bibliotecznym oznaczać mnożenie skalarne wektorów, to naszą funkcję Iloczyn_Skalarny przekształci do postaci:

```
function "*" (A, B : Wektor) return Float is
  Wynik : Float := 0.0;
begin
  if (A'First /= B'First) or (A'Last /= B'Last) then
    raise Constraint_Error;
  end if;
  for I in A'Range loop
    Wynik := Wynik + A(I)*B(I);
  end loop;
  return Wynik;
end "*";
```

W ten sposób zadeklarowaną funkcję wywołujemy w wyrażeniu, tak jak działanie. Piszemy więc

```
Iloczyn := Wektor_1 * Wektor_2;
```

Kompilator Ady rozpoznaje znaczenie operatora mnożenia z kontekstu w jakim ten operator występuje. Zauważmy, że nie może powstać niejednoznaczność, bo funkcja „*" działa w tym przypadku na argumentach typu tablicowego.

Ogólnie, jeżeli definiujemy nowy typ danych, zawsze możemy określić nowe przeciążenia operatorów takich jak „=", czy „>", przy czym operator relacyjny „=" nie musi być funkcją typu logicznego, ale jeżeli jest, to nowe przeciążenie operatora „/=" jest tworzone automatycznie.

W Adzie możliwe jest przeciążanie nie tylko operatorów, ale również innych funkcji oraz procedur. Stosowaliśmy już wielokrotnie procedury o nazwach

przeciążonych, takie jak Put i Get, które służą do wczytywania i wypisywania znaków, albo liczb. Zauważmy, że kompilator rozpoznaje na podstawie typu parametru aktualnego konkretną procedurę, którą należy wywołać. Należy podkreślić, że nie mamy tu do czynienia z przesłanianiem, które występuje wtedy, gdy ilość, kolejność i typy bazowe parametrów oraz typ wyniku są takie same, natomiast przeciążenie procedury (funkcji) może być zrealizowane wtedy, gdy deklaracja nowej procedury jest wystarczająco różna (oczywiście z wyłączeniem identyfikatora) od deklaracji procedury przeciążanej.

Na tym kończymy rozdział poświęcony podprogramom. Przykłady deklaracji procedur i funkcji oraz ich wywołań znajdzie Czytelnik w następnych rozdziałach.

Uzupełnienie

Umiejscowienie procedur. Procedury mogą znajdować się w osobnych plikach, w pakietach, albo w naszym programie. W przypadku, gdy procedura umieszczona jest w naszym programie nie musimy podawać oddzielnej deklaracji procedury[15]. Treść procedury służy również za jej deklarację.

Treść procedury umieszczamy w części deklaracyjnej programu. Podstawową zasadą dotyczącą kolejności deklaracji w Adzie jest to, że identyfikator musi musi być zdefiniowany przed jego użyciem (kompilator niczego nie zamierza się domyślać).

Procedury lokalne. Te same założenia, które dotyczą umieszczania procedur w programie, dotyczą także umieszczania procedur lokalnych. Pamiętajmy, że procedury lokalne mają automatyczny dostęp do wszystkich zmiennych otaczających statycznie te procedury co pozwala na zmniejszanie ilości przekazywanych parametrów i tym samym zwiększenie czytelności, a często i efektywnosci programów. Podprogramy lokalne **nie występują** w języku C/C++, czy Java i dlatego ich zalety są często niedoceniane przez programistów.

Zalecenie kolejności deklaracji w programie.

1. Deklaracje typów określonych przez programistę,
2. Deklaracje stałych,
3. Konkretyzacje pakietów ogólnych,
4. Deklaracje zmiennych,
5. Treści procedur.

Zalecenia stylistyczne. Identyfikator procedury powinien być czasownikiem. Na przykład wywołanie Oblicz_Srednia (Z1, X2, Srednia);. Procedura powinna zawierać komentarze wyjaśniające sposób jej wywołania i ewentualnie źródło lub opis algorytmu wg którego procedura realizuje obliczenia.

Komentarze opisujące parametry formalne można umieścić bezpośrednio po definicji parametru.

[15] Ale dobry styl programowania wymaga, by każda procedura miała swoją oddzielną deklarację, przy której w formie komentarza zawarty jest jej opis, znaczenie parametrów, ewentualne efekty uboczne itp.

Kiedy używać funkcji?

1. Jeżeli moduł ma obliczyć więcej niż jedna wartość (dana strukturalna to też jest jedna wartość), albo zmienić wartości parametrów aktualnych – nie można stosować funkcji.
2. Jeżeli moduł ma wykonywać operację wejścia/wyjścia nie stosuj funkcji. Ewentualne błędy powinny być zgłaszane poprzez wyjątki (rozdział 10).
3. Jeżeli moduł oblicza jedna wartość typu logicznego – stosuj funkcję.
4. Jeżeli moduł oblicza jedna wartość, którą zamierzamy wykorzystać jako argument wyrażenia – stosuj funkcję.
5. Jeżeli masz wątpliwości, co stosować, stosuj procedurę. Każdą funkcję można zamienić na procedurę, przy czym nazwa funkcji staje się wtedy parametrem typu out.
6. Jeżeli oba rodzaje podprogramów są równie dobre, wybierz ten rodzaj, który zapewnia Ci większą wygodę w stosowaniu.
7. Jeżeli konieczne jest stosowanie funkcji, która zmienia swoje parametry aktualne (jest to często konieczne w przypadku stosowania komunikacji z programami napisanymi w innych językach programowania np. Delphi (Teixeira i Pacheco 2002), C i C++ (citeKernighan87,Stroustrup91, w systemie MS Windows (Petzold i Yao 1997), to stosowne parametry mogą być przekazywane jako wskaźniki (rozdział 8) w formie Nazwa_Zmiennej'Access, lub Nazwa_Zmiennej'Unchecked_Access, lub też Nazwa_Zmiennej'Unrestricted_Access.

Rozdział 7

Rekurencja

Procedurę lub strukturę danych nazywamy rekurencyjną, jeżeli częściowo składa się z samej siebie lub jej definicja odwołuje się do niej samej.

Przykłady rekurencyjnych struktur danych podano w rozdziale dotyczącym typów wskaźnikowych. W tym rozdziale omawiane są natomiast procedury rekurencyjne.

Definicje rekurencyjne są często stosowane w matematyce (Arbib i inni 1981).

Oto kilka przykładów definicji rekurencyjnych:

Definicja liczb naturalnych $\mathcal{N} = \{0, 1, 2, 3, \ldots\}$

0 jest liczbą naturalną,
następnik liczby 0 jest liczbą naturalną
Przypominamy, że zbiór liczb naturalnych jest reprezentowany w Adzie przez zdefiniowany w pakiecie Standard typ Natural, który jest podtypem typu Integer, natomiast zbiór liczb całkowitych, dodatnich
$\mathcal{N}_+ = \{1, 2, 3, \ldots\}$
jest reprezentowany przez typ Positive określony w tym samym pakiecie.

Funkcja silnia (oznaczana przez !) określona dla argumentów całkowitych, nieujemnych
$0! = 1$,
$n! = n * (n-1)!\ \forall n \in \mathcal{N}_+$.

Ciąg liczb Fibonacci $n \mapsto F_n,\ n \in \mathcal{N} = \mathcal{N}_+ \cup \{0\}$

$F_0 = 0,\ F_1 = 1$,
$F_{n+1} = F_n + F_{n-1},\ n > 1$.

Potęga nieujemna, całkowita liczby rzeczywistej $x^n,\ n \in \mathcal{N}$

$x^0 = 1$,
$x^n = x * x^{n-1},\ n > 0$.

Wielomiany Legendrea $x \mapsto P_n(x),\ x \in \Re,\ n \in \mathcal{N}$

$P_0(x) = 1, \quad P_1(x) = x$,
$P_n(x) = \frac{((2n-1)P_{n-1}(x) - (n-1)P_{n-2}(x)}{n},\ n > 1$.

Użyteczność rekurencji polega na definiowaniu nieskończonych zbiorów obiektów przy pomocy skończonych wyrażeń.

Program rekurencyjny P można ogólnie zapisać jako złożenie $\wp$ instrukcji podstawowych S_i nie zawierających P i samego programu P. Można to zapisać następująco (Wirth 1989):

$P = \wp(S_i; P)$

W Adzie realizację programów rekurencyjnych umożliwiają procedury rekurencyjne, przy czym jeżeli procedura P zawiera bezpośrednio odwołanie do samej siebie, to P nazywamy *procedurą bezpośrednio rekurencyjną*, natomiast jeżeli P zawiera odwołanie do procedury Q, która zawiera bezpośrednie odwołanie do P, to P nazywamy *procedurą pośrednio rekurencyjną*.

Procedury rekurencyjne dopuszczają możliwości wykonywania nieskończonego procesu obliczeniowego i w związku z tym powstaje problem zakończenia tego procesu w skończonym czasie. Problem ten rozwiązuje się w ten sposób, że wywołanie rekurencyjne procedury P uzależnione jest od warunku *W*, który w pewnym momencie przestaje być spełniony, co zatrzymuje proces obliczeniowy. Można to zapisać w postaci:

$P = \text{if } W \text{ then } \wp(S_i; P)$

albo równoważnie

$P = \wp(S_i; \text{if } W \text{ then } P)$.

7

Prostym i skutecznym sposobem zakończenia wykonywania programu rekurencyjnego jest zastosowanie w procedurze P parametru wejściowego n i wywołanie procedury z wartością tego parametru równą $n - 1$. Jeżeli warunek *W* jest postaci $n > 0$, to następujące schematy gwarantują wykonanie programu w skończonej liczbie kroków:

$P(n) = \text{if } n > 0 \text{ then } \wp(S_i; P(n-1))$
$P(n) = \wp(S_i; \text{if } n > 0 \text{ then } P(n-1))$.

Algorytmy rekurencyjne stosuje się wtedy, gdy rozwiązywany problem, lub przetwarzane dane definiujemy rekurencyjne. Nie oznacza to jednak, że algorytm rekurencyjny jest zawsze najefektywniejszym rozwiązaniem problemu.

Dla przykładu weźmy pod uwagę funkcję silnia. Zgodnie z podanym wyżej określeniem mamy:

$i = 0, 1, 2, 3, 4, 5, \ldots,$
$\text{silnia}(i) = 1, 1, 2, 6, 24, 120, \ldots$

Obliczanie tej funkcji można zapisać w Adzie następująco:

```
1 function Silnia (N : Natural ) return Natural is
2 begin
3    if N > 0 then
4       return N*Silnia(N-1);
5    else
6       return 1;
7    end if;
8 end Silnia;
```

Podana funkcja realizuje obliczanie silni metodą rekurencyjną. Można jednak podać funkcję realizującą te obliczenia metodą iteracyjną:

```
function Silnia_Iter (N : Natural ) return Natural is
  Wynik : Natural := 1;
begin
  if N > 0 then
    for I in 1..N loop
      Wynik := I*Wynik;
    end loop;
  end if;
  return Wynik;
end Silnia_Iter;
```

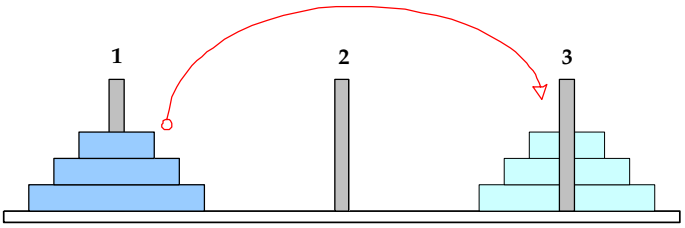
Drugim przykładem jest obliczanie wyrazów ciągu Fibonacci. Bezpośrednie zastosowanie definicji prowadzi do następującej funkcji:

```
function Fibonacci_Rek (N : Natural ) return Natural is
begin
  if N = 0 then
    return 0;
  elsif N = 1 then
    return 1;
  else
    return Fibonacci_Rek (N-1) + Fibonacci_Rek (N-2);
  end if;
end Fibonacci_Rek;
```


Obliczanie n– tego wyrazu rozpatrywanego ciągu można również zrealizować przy pomocy następującej funkcji iteracyjnej:

```
function Fibonacci (N : Natural ) return Natural is
  I, F, Y : Natural;
begin
  if N = 0 then
    F := 0;
  else
    I := 1;
    F := 1;
    Y := 0;
    while I < N loop
      I := I + 1;
      F := F + Y;
      Y := F - Y;
    end loop;
  end if;
  return F;
end Fibonacci;
```

Powstaje naturalne pytanie kiedy stosujemy rekurencję, a kiedy jej unikamy. Zauważmy, że każde rekurencyjne wywołanie procedury P wymaga pewnego obszaru pamięci rezerwowanego dla zmiennych lokalnych i na zapamiętanie aktualnego stanu obliczeń. łatwo widzieć, że każde wywołanie rekurencyjnej wersji funkcji obliczającej wyrazy ciągu Fibonacci (Fibonacci_Rek) z parametrem $n > 1$ powoduje dalsze dwa wywołania, a więc liczba wywołań rośnie wykładniczo ze wzrostem n. Dla $n = 5$ mamy 15 wywołań tej funkcji. Podobna sytuacja występuje w innych podanych przykładach.



Rysunek 7.1: Wieże Hanoi – położenie początkowe dla $n = 3$ krążków

Stąd wniosek: unikamy rekurencji, jeżeli istnieje proste rozwiązanie iteracyjne.

Ogólnie, programy w których można uniknąć rekurencji mają następujące równoważne schematy:

P = if W then (S; P)
P = (S; if W then P)

Funkcje określone wzorem:

$$F_n(x) = \begin{cases} G(x); & n = 0 \\ H(F_{n-1}(x); & n > 0 \end{cases}$$

mogą być zawsze obliczone procedurą iteracyjną. Takie funkcje omawiane były w podanych przykładach.

Istnieją jednak problemy, w których rekurencja daje najprostsze rozwiązanie. Dynamiczne struktury danych omawiane w rozdziale 8 nie mogą być zdefiniowane bez użycia rekurencji.

Poniżej omawiamy dwa problemy, dla których rozwiązanie rekurencyjne jest najbardziej odpowiednie.

7.1 Wieże Hanoi

Dane są trzy pionowe pręty 1, 2, 3 i n krążków o różnej średnicy nałożonych na pręt 1 tak, że krążek o największej średnicy leży na samym dole, a następne krążki leżą jeden na drugim zgodnie z malejącą średnicą krążków. W ten sposób na pręcie 1 utworzona jest zwężająca się ku górze wieża. Sytuację tę dla $n = 3$ ilustruje rysunek 7.1.

Nasze zadanie polega na podaniu algorytmu pozwalającego na przełożeniu wieży na jeden wybrany pręt (2 albo 3), przy czym muszą być spełnione następujące warunki:

1. W jednym kroku można przekładać tylko jeden krążek.
2. Nie wolno kłaść krążka o większej średnicy na krążku o mniejszej średnicy.
3. Trzeci pręt może służyć jako pomocniczy magazyn krążków, przy czym przy nakładaniu krążków na ten pręt obowiązują reguły 1 i 2.

Załóżmy, że przesuwamy wieżę z pręta 1 na 3. Można to zapisać następująco:

PrzesunWieze (n, 1, 2, 3),

co oznacza, że przesuwamy wieżę o wysokości n z pręta 1 na 3 korzystając z pręta pomocniczego 2. Prawidłowe i proste rozwiązanie tego typu problemu umożliwia obserwacja, że ważny jest dolny krążek na pręcie 1, a nie górny.

Instrukcję PrzesunWieze (n, 1, 2, 3) można wtedy rozłożyć na trzy następujące zadania:

1. PrzesunWieze (n − 1, 1, 3, 2);
2. Przenies krazek z preta 1 na 3;
3. PrzesunWieze (n − 1, 2, 1, 3)

Oczywiście w sytuacji, gdy $n = 1$ algorytm ten nie działa i w związku z tym musimy dopisać:

Nic nie rob, gdy n = 1

Możemy teraz napisać program realizujący opisany algorytm

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure WiezeHanoi is
5     subtype Krazki is Natural range 0 .. 64;
6     subtype Prety is Positive range 1..3;
7     Ile : Krazki;
8
9     procedure Przenies (X, Y : in Prety ) is
10    begin
11       New_Line;
12       Put (" ");
13       Put(X, 2);
14       Put(" na ");
15       Put(Y, 2);
16    end Przenies;
17
18    procedure PrzesunWieze (N : in Krazki; A, B, C : in Prety ) is
19    begin
20       if N > 0 then
21                PrzesunWieze (N-1,A,C,B);
22          Przenies (A,C);
23                PrzesunWieze (N-1,B,A,C);
24       end if;
25    end PrzesunWieze;
26
27 begin
28    New_Line;
29    Put ("Podaj ilosc krazkow ");
30    Get (Ile);
31    New_Line;
32    Put (" Wyniki dla N = ");
33    Put(Ile);
```

7

```
34    New_Line;
35    PrzesunWieze (Ile,1, 2, 3);
36 end WiezeHanoi;
```

Procedura pomocnicza Przenies wypisuje numer pręta, z którego pobierany jest krążek, napis na i numer pręta na który nakładany jest ten krążek.

Oto wyniki działania programu dla $Ile = n = 3$:

```
1 na 3
1 na 2
3 na 2
1 na 3
2 na 1
2 na 3
1 na 3
```

Drugim, pouczajacym przykładem są

7.2 Permutacje

Problem polega na wypisaniu wszystkich możliwych permutacji n różnych obiektów $A(1) \ldots A(n)$. Nazywając taką operację Permutuj(n), algorytm można sformułować następująco:

Najpierw pozostaw A(n) na swoim miejscu i wykonaj Permutuj(n−1), a następnie powtórz ten proces po zamianie A(n) z A(i) dla $i = 1$. Kontynuuj te operacje dla wszystkich $i = 2, \ldots, n-1$. Poniżej przytaczamy program, który realizuje ten algorytm.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2
3  procedure Permutacje_Rek is
4    -- Program wypisuje permutacje zbioru wielkich liter - wersja
5    -- rekurencyjna
6    subtype Wielkie_Litery is Character range 'A' .. 'Z';
7    Ile_Liter : constant Positive := 3;
8    type Zbior_Liter is array (Positive range <>) of Wielkie_Litery;
9    Nasz_Zbior : Zbior_Liter (1 .. Ile_Liter);
10
11   procedure Wczytaj_Zbior (Zbior : out Zbior_Liter ) is
12     I : Positive;
13     Znak : Wielkie_Litery;
14   begin
15     Put ("Podaj wielkie litery ");
16     New_Line;
17     I := 1;
18     loop
19       Get (Znak);
20       if Znak not in Wielkie_Litery then
21         Put ("Prawidlowy znak to wielka litera. Wpisz nastepny");
22       else
23         Zbior(I) := Znak;
24         I := I + 1;
```

```
25          end if;
26          exit when (I > Zbior'Last);
27       end loop;
28    end Wczytaj_Zbior;
29
30    procedure Wypisz_Zbior (Zbior : in Zbior_Liter ) is
31    begin
32       for I in Zbior'range loop
33          Put (Zbior(I));
34          Put (" ");
35       end loop;
36       New_Line;
37    end Wypisz_Zbior;
38
39    procedure Zamien (Zbior : in out Zbior_Liter;
40                      I, K : in Positive ) is
41       Temp : Character;
42    begin
43       Temp := Zbior(I);
44       Zbior(I) := Zbior(K);
45       Zbior(K) := Temp;
46    end Zamien;
47    procedure Permutuj (Zbior : in Zbior_Liter;
48                        K : in Positive ) is
49       Zbior_Temp : Zbior_Liter (1 .. Zbior'Last);
50    begin
51       Zbior_Temp := Zbior;
52       if K = Zbior_Temp'Last then
53          Wypisz_Zbior (Zbior_Temp);
54       else
55          for I in K..Zbior_Temp'Last loop
56             Zamien (Zbior_Temp, I, K);
57             Permutuj (Zbior_Temp, K+1);
58          end loop;
59       end if;
60    end Permutuj;
61 begin
62    Wczytaj_Zbior (Nasz_Zbior);
63    New_Line;
64    Put ("Oto zbior wejsciowy: ");
65    New_Line;
66    Wypisz_Zbior (Nasz_Zbior);
67    New_Line;
68    Put ("To sa jego permutacje: ");
69    New_Line;
70    Permutuj (Nasz_Zbior,1);
71 end Permutacje_Rek;
```

Dla $n = 3$ otrzymano następujace wyniki:

```
Oto zbior wejsciowy:
X Y Z

To sa jego permutacje:
X Y Z
```

```
X Z Y
Y X Z
Y Z X
Z X Y
Z Y X
```

Możliwe jest rozwiązanie problemu permutacji metodą iteracyjną, ale algorytm iteracyjny jest bardziej skomplikowany i mniej czytelny niż prezentowana tu wersja rekurencyjna.

Na tym kończymy omawianie rekurencji. Znacznie pełniejsze omówienie różnych aspektów algorytmów rekurencyjnych oraz interesujące przykłady ich zastosowania znaleźć można w książkach (Alagić i Arbib 1978, Arbib i inni 1981, Cormen, Leiserson i Rivest 1997, Harel 1992, Ross i Wright 1999, Wirth 1989).

7.3 Inny przykład rekurencji

Przypuśćmy, że chcemy przeczytać z terminala (albo i z pliku) linię o dowolnej długości. Przypuśćmy też, że nie chcemy korzystać ze standardowego pakietu Ada.Strings.Unbounded.Text_IO, tylko w pewnym miejscu programu chcemy utworzyć dynamiczny napis np. w następujący sposób:


```
declare
   Napis : String := Get_Line;
begin
   ...
end;
```

Korzystając ze standardowej funkcji Ada.Text_IO.Get_Line muimy określić maksymalną długość linii w związku z tym alokujemy albo ogromny napis „na zapas”, albo też możemy napisać następującą procedurę rekurencyjną:

```
1  function Get_Line return String is
2      Buffer : String (1..500); -- lub też napis innej długości
3                               -- podyktowany względami optymalizacyjnymi
4      Last : Natural;
5  begin
6      Ada.Text_IO.Get_Line (Buffer, Last);
7      if Last < Buffer'Last then
8          return Buffer (1..Last);
9      else
10         return Buffer & Get_Line;
11     end if;
12 end Get_Line;
```

Rozdział 8

Wskaźniki i dynamiczne struktury danych

Tablica, rekord i zbiór są statycznymi strukturami danych. Oznacza to, że zmienne tych typów strukturalnych zachowują tę samą strukturę przez cały czas istnienia tych zmiennych. W wielu zastosowaniach wygodniej jest używać danych, które nie tylko zmieniają swoje wartości, ale również skład, wymiar i strukturę. Typowymi przykładami takich struktur danych są listy i drzewa, które rosną i kurczą się dynamicznie. Do konstrukcji dynamicznych struktur danych służy pojęcie *wskaźników*.

W Adzie istnieją dwa rodzaje typów wskaźnikowych: ograniczone i ogólne. *Zmienne typów wskaźnikowych ograniczonych* mogą wskazywać tylko obiekty utworzone dynamicznie, natomiast zmienne *typów wskaźnikowych ogólnych* wskazują na zadeklarowane obiekty lub podprogramy.

8.1 Typy wskaźnikowe ograniczone

Każda złożona struktura danych składa się ostatecznie z elementów, których struktura jest statyczna. Wskaźniki, tzn. wartości typów wskaźnikowych, nie są strukturalne, ale służą do ustalenia zależności pomiędzy tymi elementami statycznymi, zwykle nazywanymi węzłami. Mówi się, że wskaźniki wskazują, albo łączą elementy. Często wraz ze zmianą struktury danych niezbędna jest zmiana liczby elementów wchodzących w skład tej struktury i powiązanych przez wskaźniki. Poznany wcześniej mechanizm tworzenia zmiennych przez deklarowanie ich w części deklaracyjnej programu, nie pozwala na zmianę tej liczby w trakcie realizacji programu. Zmienne wskaźnikowe służą do odwołań do zmiennych, które nie są deklarowane jawnie w deklaracjach zmiennych i dlatego nie można się do nich odwoływać za pomocą identyfikatorów. Zmienne do których odwołujemy się poprzez wskaźniki są więc anonimowe i nazywane są *zmiennymi dynamicznymi*. Jak wiadomo, tworzenie zmiennych statycznych jest uwidocznione w statycznej strukturze programu przez jawną dekla-

rację zmiennych, które następnie oznaczane są przez swoje identyfikatory, przy czym każda zmienna statyczna istnieje podczas całego wykonywania części programu dla której jest lokalna. Zmienna dynamiczna jest natomiast tworzona przez użycie alokatora przydzielającego pamięć dla tej zmiennej, co zapisujemy:

```
Zmienna_Wskaznikowa := new Typ_Wskazywany;
```

Wskaźniki są użytecznym narzędziem dlatego, że mogą wskazywać zmienne, których składowymi są również wskaźniki. Podobny mechanizm występuje w procedurach wywołujących procedury, co prowadzi do pojęcia rekurencji. Wskaźniki pozwalają na tworzenie rekurencyjnych struktur danych, takich jak wspomniane poprzednio listy i drzewa. Podobnie jak każde rekurencyjne wywołanie procedury musi być zakończone w skończonym czasie, również każda rekurencyjna struktura danych musi być skończona[1].

W Adzie wskaźniki ograniczone nie mogą wskazywać na dowolne zmienne. Typ zmiennej wskazywanej musi być podany w deklaracji typu wskaźnikowego i mówimy w związku z tym, że *typ wskaźnikowy* jest związany ze *wskazywanym typem* obiektu, nazywanym czasami *typem bazowym*. Rozważmy deklarację

```
type Wezel;
type Wskaznik is access Wezel;
type Wezel is
   record
      Klucz : Typ_Klucz;
      Nastepny : Wskaznik;
      Dane : Informacje;
   end record;
```

8

Deklaracja ta określa nowy typ Wskaznik będący zbiorem wartości wskazujących dane typu Wezel. Wskaznik nazywamy typem wskaźnikowym związanym z typem Wezel. Każdy typ wskaźnikowy zawiera wartość null, która nic nie wskazuje, czyli nie wskazuje na żadną daną. Mechanizm wskazywania ilustruje rysunek 8.1, na którym zmienna wskaźnikowa Wsk_Listy wskazuje węzeł listy liniowej, który jest reprezentowany przez typ rekordowy Wezel. Typ ten ma trzy składowe: Klucz zawierający informacje służące do identyfikacji elementu listy, Nastepny, który wskazuje następny węzeł listy i Dane zawierające właściwe informacje przechowywane w węźle. Zauważmy, że podana deklaracja jest rekurencyjna, ponieważ zawiera składową typu wskaźnikowego związanego z typem, którego jest składową. Jeżeli dana jest deklaracja zmiennej:

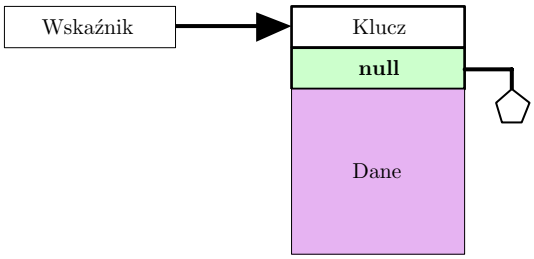
```
Wsk_Listy : Wskaznik;
```

to mówimy, że Wsk_Listy jest zmienną wskaźnikową związaną z typem Wezel. Zmienna ta wskazuje na obszar pamięci, w którym znajdują się dane typu Wezel. Początkowo zmienna wskaźnikowa ma wartość null, przy czym nie jest konieczne jawne jej inicjowanie, ponieważ wszystkie zmienne wskaźnikowe otrzymują tę wartość przy deklaracji. Jeżeli chcemy przydzielić obszar pamięci dla danych typu Wezel, to robimy to pisząc:

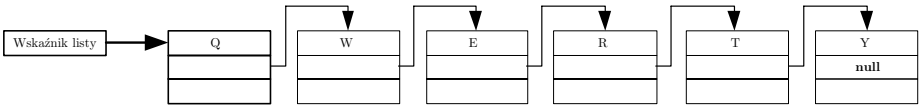
```
Wsk_Listy := new Wezel;
```

Wykonanie tej instrukcji przydziela odpowiedni obszar pamięci, który wskazywany jest przez zmienną Wsk_Listy (patrz rysunek 8.1). Po przydzieleniu pamięci

[1] To w żadnym wypadku nie oznacza, że w Adzie nie można tworzyć np. drzew cyklicznych!



Rysunek 8.1: Mechanizm wskazywania



Rysunek 8.2: Schemat sześcioelementowej listy liniowej

na dane typu Wezel chcemy zapewne nadać wartości, które będą przechowywane w nowoutworzonym węźle, przy czym składowa typu Wskaznik w momencie alokowania pamięci otrzymuje wartość domyślną null. Innym składowym możemy nadać wartości przy pomocy instrukcji:

```
Wsk_Listy.Klucz := Wartosc_Klucz;
Wsk_Listy.Dane := Wartosc_Dane;
```

Zwróćmy uwagę na to, że dane w węźle są anonimowe i nadajemy im wartości przy pomocy odpowiedniej zmiennej wskaźnikowej. Wartości składowych węzła mogą być nadane w momencie wywołania alokatora. Można to zrobić pisząc:

```
Wsk_Listy := new Wezel'(Wartosc_Klucz, null, Wartosc_Dane);
```

W tym przypadku wartość początkowa jest agregatem kwalifikowanym i musimy jawnie nadać wartości wszystkim składowym rekordu, nawet wtedy, gdy niektóre mają wartości domyślne. Jak pamiętamy, w Adzie możemy inicjować zmienne przy ich deklaracji. Dotyczy to również zmiennych wskaźnikowych. Poprawna jest więc deklaracja

```
Wsk_Listy : Wskaznik := new Wezel'(Wartosc_Klucz, null, Wartosc_Dane);
```

Dalej zajmiemy się najbardziej typowymi zastosowaniem wskaźników ograniczonych, a więc omówimy ich zastosowanie do przetwarzania list liniowych i tworzenia drzew binarnych. Należy dodać, że tutaj opisujemy jedynie najbardziej elementarne pojęcia i metody związane z dynamicznymi strukturami danych. Doskonałe omówienie tego tematu można znaleźć w klasycznej książce Niklausa Wirtha (Wirth 1989), przy czym zwracamy uwagę Czytelnika, że w tej książce algorytmy zapisywano w języku Pascal. Bardziej zaawansowany Czytelnik, pragnący zapoznać się ze współczesnym podejściem opartym o pojęcia i metody programowania obiektowego i wyrażonym w naszym ulubionym języku Ada może skorzystać z podręcznika (Beidler 1997).

Najprostszymi rekurencyjnymi strukturami danych są *listy jednokierunkowe* nazywane również *listami liniowymi*. Charakteryzują się one tym, że zawie-

rają węzły, przy czym każdy węzeł ma jeden element, który z kolei sam jest wskaźnikiem (do tego samego typu węzła).

Na rysunku 8.2. pokazano schemat listy liniowej zawierającej sześć węzłów ze zmienną wskaźnikową Wsk_Listy : Wskaznik, która wskazuje na pierwszy węzeł.

Pierwszą operacją związaną z listami jest tworzenie listy. Podany niżej program tworzy listę i wypisuje jej zawartość.

```
1  with Ada.Text_Io;
2  use Ada.Text_Io;
3  procedure Lista_Liniowa is
4    subtype Informacje is Character range 'A'..'Z';
5    type Wezel;
6    type Wskaznik is access Wezel;
7    type Wezel is
8    record
9      Klucz : Informacje;
10     Nastepny : Wskaznik;
11   end record;
12   Znak : Character; -- czytany z klawiatury
13   Element : Informacje; -- szukany, lub wstawiany
14   Nowy_Element : Informacje; -- nowowstawiany
15   Wsk_Listy : Wskaznik; -- wskazuje na pierwszy węzeł
16   Wsk_Ostatni : Wskaznik; -- wskazuje na ostatni węzeł
17   Wsk_Nowy : Wskaznik; -- uzywany do tworzenia nowych węzłów
18   Wsk_Aktualny : Wskaznik; -- wskazuje znaleziony węzeł
19   Wsk_Poprzedni : Wskaznik; -- wskazuje poprzedni węzeł
20   procedure Wypisz_Klucz (Wsk : in Wskaznik ) is
21   begin
22     if Wsk /= null then
23       Put (Wsk.Klucz);
24       Put (" ");
25     else
26       Put (" wskaznik ma wartosc null");
27     end if;
28   end Wypisz_Klucz;
29   procedure Wypisz_Liste (Wsk : in Wskaznik ) is
30     Aktualny : Wskaznik;
31   begin
32     Aktualny := Wsk;
33     New_Line;
34     Put ("Oto lista : ");
35     while Aktualny/= null loop
36       Wypisz_Klucz (Aktualny);
37       Aktualny:= Aktualny.Nastepny;
38     end loop;
39   end Wypisz_Liste;
40 begin
41   Put ("Wprowadz znaki ");
42   New_Line;
43   Get (Znak);
44   if Znak in Informacje then
45     Wsk_Nowy := new Wezel; -- Pierwszy węzeł listy
46     Wsk_Nowy.Klucz := Znak;
```

```
47        Wsk_Nowy.Nastepny := null;
48        Wsk_Listy := Wsk_Nowy;
49        Wsk_Ostatni := Wsk_Nowy;
50        loop
51          Get (Znak);
52          exit when Znak not in Informacje;
53          Wsk_Nowy := new Wezel; -- Następne węzły
54          Wsk_Nowy.Klucz := Znak;
55          Wsk_Nowy.Nastepny := null;
56          Wsk_Ostatni.Nastepny := Wsk_Nowy;
57          Wsk_Ostatni := Wsk_Nowy;
58        end loop;
59        Wypisz_Liste (Wsk_Listy);
60        New_Line;
61        Put ("Klucz w ostatnim wezle = ");
62        Wypisz_Klucz (Wsk_Ostatni);
63        New_Line;
64        Put ("Klucz w pierwszym wezle = ");
65        Wypisz_Klucz (Wsk_Listy);
66        -- Inne operacje na utworzonej liście
67      else
68        Put ("Lista pusta");
69        -- Operacje na pustej liście
70      end if;
71  end Lista_Liniowa;
```

Zwróćmy uwagę na to, że zmienne Wsk_Listy, Wsk_Ostatni wskazują odpowiednio pierwszy i ostatni węzeł listy, a kluczami poszczególnych węzłów są wielkie litery, przy czym węzły nie zawierają innych danych oprócz kluczy i wskaźników do następnych węzłów. Łatwo zauważyć, że tworzenie listy kończy się, gdy wprowadzimy inny znak niż wielka litera. Jeżeli wprowadzimy kolejno litery Q, W, E, R, T, Y i zakończymy tę sekwencję np. znakiem odstępu, to otrzymamy wyniki:

```
Oto lista : Q W E R T Y
Klucz w ostatnim wezle = Y
Klucz w pierwszym wezle = Q
```

Po skonstruowaniu listy przez kolejne wstawianie węzłów, zwykle trzeba przeszukać listę w celu znalezienia klucza o podanej przez nas wartości. W tym celu możemy użyć dwóch procedur:

```
procedure Czytaj_Klucz (Zn : in out Character;
                        Kl : out Informacje ) is
begin
  loop
    New_Line;
    Put ("Podaj klucz ");
    Get (Zn);
    exit when Zn in Informacje;
    Put ("To ma byc wielka litera }");
  end loop;
  Kl := Zn;
end Czytaj_Klucz;
```

```
procedure Szukaj (Wsk : in Wskaznik;
                  Element : in Informacje;
                  Aktualny : out Wskaznik;
                  Poprzedni : out Wskaznik ) is
begin
  Aktualny := Wsk;
  Poprzedni := null;
  while (Aktualny /= null) and then (Aktualny.Klucz /= Element) loop
    Poprzedni := Aktualny;
    Aktualny := Aktualny.Nastepny;
  end loop;
end Szukaj;
```

Użyto ich w naszym programie następująco:

```
Czytaj_Klucz (Znak, Element);
Szukaj (Wsk_Listy, Element, Wsk_Aktualny, Wsk_Poprzedni);
Put ("Klucz znaleziony = ");
Wypisz_Klucz (Wsk_Aktualny);
New_Line;
Put ("Klucz poprzedni = ");
Wypisz_Klucz (Wsk_Poprzedni);
```

Zwróćmy uwagę na to, że w procedurze Szukaj korzystamy z operatora and then, który wartościuje prawy argument, tylko wtedy, gdy lewy argument ma wartość True. Gdybyśmy zastosowali zwykły operator koniunkcji, to mogłaby zostać wyznaczona składowa Aktualny.Klucz /= Element, w przypadku gdy Aktualny = null, co jest niedopuszczalne[2],

8

Wyniki przeszukiwania poprzednio skonstruowanej listy są następujące:

1. Podaj klucz Q
   Klucz znaleziony = Q
   Klucz poprzedni = wskaznik ma wartosc null,
2. Podaj klucz Y
   Klucz znaleziony = Y
   Klucz poprzedni = T,
3. Podaj klucz A
   Klucz znaleziony = wskaznik ma wartosc null
   Klucz poprzedni = Y
4. Lista pusta Podaj klucz A
   Klucz znaleziony = wskaznik ma wartosc null
   Klucz poprzedni = wskaznik ma wartosc null

W trzecim przypadku nie znaleziono właściwego elementu i zmienna Wsk_Aktualny przyjęła wartość null, natomiast zmienna Wsk_Poprzedni wskazuje na ostatni węzeł, ponieważ przeszukujemy listę od strony lewej do prawej. Ostatni przypadek pokazuje wyniki przeszukiwania pustej listy.

[2] W takim wypadku, w czasie wykonania programu, byłby zgłoszony wyjątek CONSTRAINT_ERROR.

Załóżmy, że utworzyliśmy listę i chcemy do niej wstawić po elemencie wskazywanym zmienną wskaźnikową Wsk_Aktualny, nowy element. Możemy w tym celu użyć następującej procedury:

```
1  procedure Wstaw_Po_Wezle (Wsk : in Wskaznik;
2                            Kl : in Informacje;
3                            Kl_Nowy : in Informacje ) is
4    Poprzedni, Nowy, Aktualny : Wskaznik;
5  begin
6    Szukaj (Wsk, Kl, Aktualny, Poprzedni);
7    if Aktualny /= null then
8      Nowy := new Wezel;
9      Nowy.Klucz := Kl_Nowy;
10     Nowy.Nastepny := Aktualny.Nastepny;
11     Aktualny.Nastepny := Nowy;
12   end if;
13 end Wstaw_Po_Wezle;
```

Wywołanie tej procedury:

```
Wstaw_Po_Wezle (Wsk_Listy, Element, Nowy_Element);
```

dało w naszym programie następujące wyniki:

1. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla po ktorym wstawiamy nowy wezel
   Podaj klucz Q
   Oto lista : Q O W E R T Y

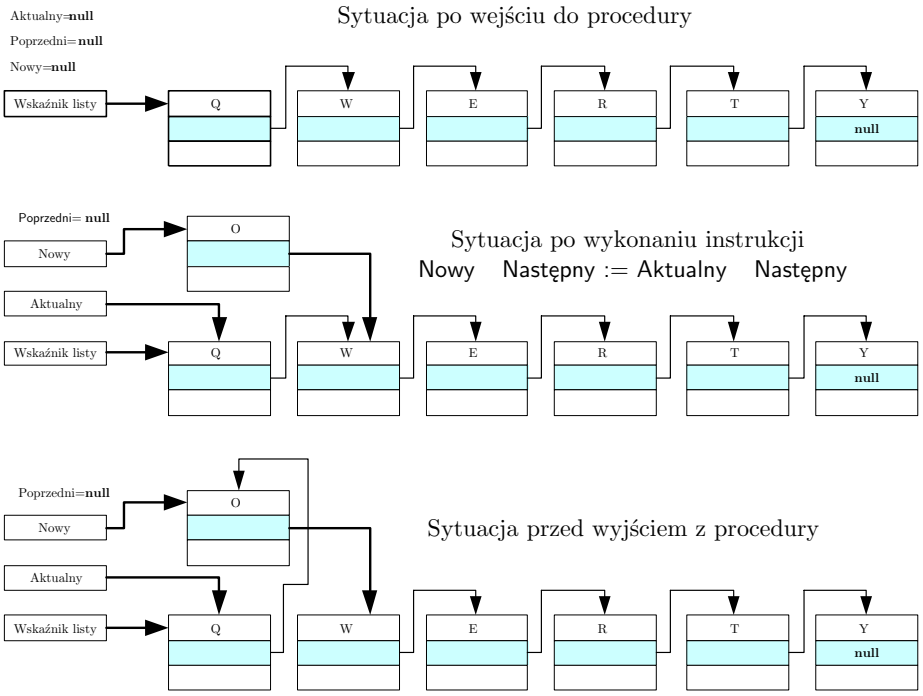
2. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla po ktorym wstawiamy nowy wezel
   Podaj klucz Y
   Oto lista : Q W E R T Y O

3. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla po ktorym wstawiamy nowy wezel
   Podaj klucz O
   Oto lista : Q W E R T Y

W ostatnim teście nie znaleziono węzła z odpowiednim kluczem i dlatego lista pozostała niezmieniona. Sytuację przed i po wstawieniu węzła z kluczem O, po węźle z kluczem Q (przypadek 1) ilustruje rysunek 8.3.

Jeżeli chcemy wstawić węzeł przed wybrany węzeł, to możemy użyć następującej procedury:

```
1  procedure Wstaw_Przed_Wezel (Wsk : in out Wskaznik;
2                               Kl : in Informacje;
3                               Kl_Nowy : in Informacje) is
4    Poprzedni, Nowy : Wskaznik;
5    Aktualny : Wskaznik;
```



Rysunek 8.3: Wstawianie nowego węzła po wybranym węźle

8

```
6   begin
7     Szukaj (Wsk, Kl, Aktualny, Poprzedni);
8     if Aktualny /= null then
9       Nowy := new Wezel;
10      Nowy.Klucz := Kl_Nowy;
11      Nowy.Nastepny := Aktualny;
12      if Poprzedni /= null then
13        Poprzedni.Nastepny := Nowy;
14      else
15        Wsk := Nowy;
16      end if;
17    end if;
18  end Wstaw_Przed_Wezel;
```

Przeprowadzone testy dały wyniki:

1. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla przed ktory wstawiamy nowy wezel
   Podaj klucz R
   Oto lista : Q W E O R T Y

2. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla przed ktory wstawiamy nowy wezel
   Podaj klucz Q
   Oto lista : O Q W E R O T Y

```
3. Wprowadz klucz nowego wezla
   Podaj klucz O
   Wprowadz klucz wezla przed ktory wstawiamy nowy wezel
   Podaj klucz O
   Oto lista : Q W E R T Y
```

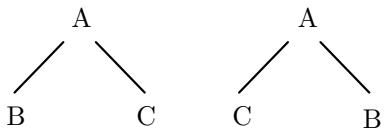
W ostatnim przypadku nie wstawiono nowego węzła, bo w liście nie ma węzła z kluczem O.

Usuwanie węzła z listy realizuje

```
1  procedure Usun (Wsk : out Wskaznik;
2                  Kl : in Informacje ) is
3    Aktualny : Wskaznik;
4    Poprzedni : Wskaznik;
5  begin
6    Szukaj (Wsk, Kl, Aktualny, Poprzedni);
7    if Aktualny /= null then
8      if Poprzedni /= null then
9        Poprzedni.Nastepny := Aktualny.Nastepny;
10     else
11       Wsk := Aktualny.Nastepny;
12     end if;
13   end if;
14 end Usun;
```

Jeżeli chcemy wstawić węzeł na początek listy, albo na jej koniec, to możemy użyć następujących procedur:

```
1  procedure Wstaw_Na_Poczatek (Wsk : in out Wskaznik;
2                               Kl : in Informacje ) is
3    Nowy : Wskaznik;
4  begin
5    Nowy := new Wezel;
6    Nowy.Klucz := Kl;
7    Nowy.Nastepny := Wsk;
8    Wsk := Nowy;
9  end Wstaw_Na_Poczatek;
10
11 procedure Wstaw_Na_Koniec (Wsk : in out Wskaznik;
12                            Kl : in Informacje ) is
13   Ostatni : Wskaznik;
14   Nowy : Wskaznik;
15 begin
16   Nowy := new Wezel;
17   Nowy.Klucz := Kl;
18   Nowy.Nastepny := null;
19   if Wsk /= null then
20     Ostatni := Wsk;
21     while Ostatni.Nastepny /= null loop
22       Ostatni := Ostatni.Nastepny;
23     end loop;
24     Ostatni.Nastepny := Nowy;
25   else
26     Wsk := Nowy;
```



Rysunek 8.4: Dwa różne drzewa binarne

```
27       end if;
28    end Wstaw_Na_Koniec;
```

Bardziej ogólnymi, dynamicznymi strukturami danych są struktury drzewiaste. Najpierw podamy kilka definicji (Wirth 1989).

Struktura drzewiasta, albo *drzewo* o typie podstawowym T jest

1. Strukturą pustą, albo
2. Węzłem typu T ze skończoną ilością dowiązanych, rozłącznych struktur drzewiastych o tym samym typie podstawowym, nazywanych poddrzewami.

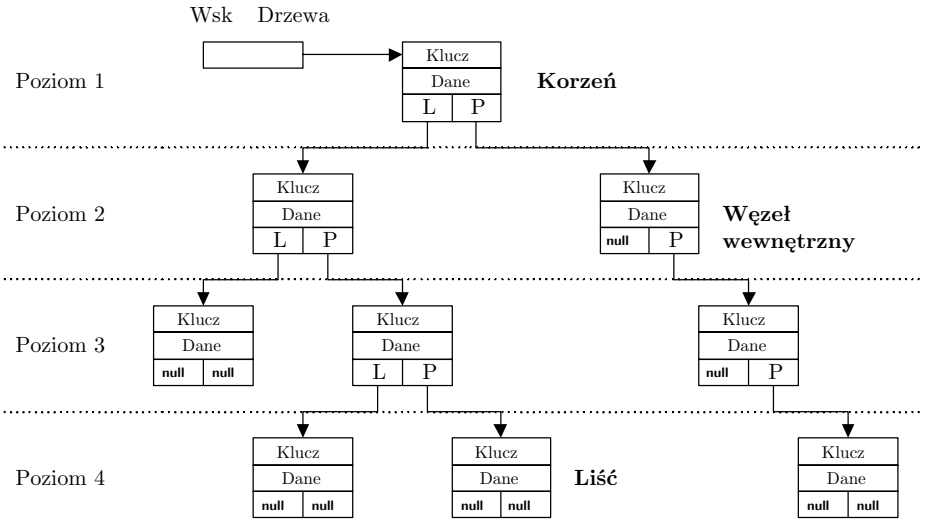
Korzeniem nazywamy najwyższy węzeł drzewa. *Drzewem uporządkowanym* jest drzewo, w którym gałęzie każdego węzła są uporządkowane. Zgodnie z tym, drzewa pokazane na rysunku 8.4 są różne. Węzeł Y, który znajduje się bezpośrednio poniżej węzła X nazywamy (bezpośrednim) potomkiem X i odpowiednio X jest (bezpośrednim) przodkiem Y. Jeżeli X jest na poziomie i, to Y jest na poziomie $i+1$. Na naszym rysunku A jest przodkiem (rodzicem) B i C, natomiast B i C są jego potomkami (dziećmi). Korzeń jest węzłem, który nie ma przodków i jest na poziomie 1, natomiast *liściem* nazywamy węzeł, który nie ma potomków. Węzły, które nie są liśćmi i korzeniem nazywamy *węzłami wewnętrznymi* (drzewa na rysunku nie mają węzłów wewnętrznych).

Uporządkowanym drzewem binarnym nazywamy drzewo uporządkowane składające się ze skończonej ilości węzłów, przy czym może być puste, albo posiada korzeń z dwoma rozłącznymi drzewami binarnymi, nazywanymi *lewym* i *prawym poddrzewem* korzenia.

Dalej, gdy mówimy o drzewach, mamy na myśli wyłącznie uporządkowane drzewa binarne.

Drzewem dokładnie wyważonym nazywamy takie drzewo, w którym na wszystkich poziomach utworzono maksymalną możliwą ilość węzłów. Drzewo pokazane na rysunku 8.5 nie jest dokładnie wyważone, ponieważ poziom 3 nie ma maksymalnej możliwej ilości węzłów, która wynosi $2^2 = 4$. Nietrudno zadeklarować typy danych, służące do reprezentowania węzłów drzew binarnych:

```
type Wezel;
type Wskaznik is access Wezel;
type Wezel is
  record
      Klucz : Informacje;
      Lewy, Prawy : Wskaznik;
    end record;
```



Rysunek 8.5: Drzewo binarne niedokładnie wyważone

Deklaracje te są bardzo podobne do deklaracji stosowanych w programie przetwarzającym listę liniową. Istotna różnica polega na tym, że teraz typ Wezel ma dwie składowe typu Wskaznik służące do utworzenia połączeń z lewym i prawym poddrzewem.

Poniżej podano tekst programu, który tworzy drzewo binarne dokładnie wyważone o zadanej liczbie węzłów i wypisuje wartości kluczy poszczególnych węzłów tak, że dobrze widać strukturę drzewa.

8

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Drzewo_Binarne is
5
6     subtype Informacje is Positive range 1..20;
7     type Wezel;
8     type Wskaznik is access Wezel;
9     type Wezel is
10    record
11       Klucz : Informacje;
12       Lewy, Prawy : Wskaznik;
13    end record;
14    Ilosc_Wezlow : Natural;
15    Korzen : Wskaznik;
16    Wciecie : Positive := 1;
17
18    function Drzewo (N : in Natural) return Wskaznik is
19       Nowy : Wskaznik;
20       L, P : Natural;
21       X : Informacje;
22       -- Funkcja służy do budowy drzewa binarnego dokładnie
23       -- wyważonego składającego się z N węzłów,
24       -- L - ilość lewych węzłów, P - ilość prawych węzłów
```

```
25   begin
26     if N = 0 then
27       return null;
28     else
29       L := N / 2;
30       P := N - L -1;
31       Put ("Klucz = ");
32       Get (X);
33       Nowy := new Wezel;
34       Nowy.Klucz := X;
35       Nowy.Lewy := Drzewo (L); -- Rekurencyjne tworzenie lewego drzewa
36       Nowy.Prawy := Drzewo (P); -- Rekurencyjne tworzenie prawego drzewa
37       return Nowy;
38     end if;
39   end Drzewo;
40
41   procedure Wypisz_Drzewo (T : in Wskaznik; W : in Natural) is
42     -- Procedura wypisuje drzewo z wcięciem = W
43   begin
44     if T /= null then
45       Wypisz_Drzewo (T.Lewy, W + 1);
46       for I in 1 .. W loop
47         Put ("   ");
48       end loop;
49       Put (T.Klucz, 2);
50       New_Line;
51       Wypisz_Drzewo (T.Prawy, W + 1);
52     end if;
53   end Wypisz_Drzewo;
54
55 begin
56   Put ("Podaj liczbe wezlow ");
57   Get (Ilosc_Wezlow);
58   Korzen := Drzewo (Ilosc_Wezlow);
59   Put ("Drzewo binarne dokladnie wywazone. ");
60   Put (" Ilosc wezlow = ");
61   Put (Ilosc_Wezlow, 2);
62   New_Line;
63   Wypisz_Drzewo (Korzen, Wciecie);
64 end Drzewo_Binarne;
```

Warto zwrócić uwagę na to, że funkcja Drzewo jest funkcją rekurencyjną, co zapewnia prostą i elegancką realizację algorytmu generującego odpowiednie drzewo. Wyniki działania naszego programu w przypadku, gdy Ilosc_Wezlow = 9 i gdy wpisywano kolejno wartości kluczy 1, 2, 3, 4, 5, 6, 7, 8, 9 są następujące:

```
Drzewo binarne dokladnie wywazone. Ilosc wezlow = 9
```

```
            4
        3
    2
        5
1
            8
        7
    6
        9
```

Dalsze informacje na temat dynamicznych struktur danych i algorytmach realizujących różne operacje na tych strukturach można znaleźć we wspomnianych już książkach Wirtha i Beidlera (Wirth 1989, Beidler 1997).

Z deklaracji

```
Zmienna_Wskaznikowa := new Typ_Wskazywany;
```

nie wynika, że typ wskazywany musi być typem rekordowym, chociaż taki typ jest najdogodniejszy przy tworzeniu dynamicznych struktur danych. Wskaźniki mogą odnosić się do dowolnych typów danych, również do typów wskaźnikowych. Poniżej podano krótki program ilustrujący deklarację i użycie operacji podstawienia w przypadku zmiennych wskaźnikowych związanych z typem Integer. Zwrócmy uwagę na sposób nadawania wartości danym wskazywanym przy ich tworzeniu instrukcją new, oraz na odwoływanie się do wartości danych wskazywanych np. Wsk1.all.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3
4  procedure Wskazniki_Integer is
5     type Wsk_Integer is access Integer;
6     Wsk1, Wsk2 : Wsk_Integer;
7  begin
8     Wsk1 := new Integer'(0);
9     Wsk2 := new Integer'(100);
10    Put (Wsk1.all, 3); New_Line;
11    Put (Wsk2.all, 3); New_Line;
12    Wsk2.all := Wsk1.all;
13    Put (Wsk1.all, 3); New_Line;
14    Put (Wsk2.all, 3); New_Line;
15    Wsk1.all := 5;
16    Wsk2.all := 25;
17    Put (Wsk1.all, 3); New_Line;
18    Put (Wsk2.all, 3); New_Line;
19 end Wskazniki_Integer;
```

Zgodnie z naszymi oczekiwaniami program wypisał następujące wyniki:

```
  0
100
  0
  0
  5
 25
```

Drugi program ilustruje najprostsze operacje wykonywane na danych wskaźnikowych związanych odpowiednio z typami standardowymi Integer, Float i Character oraz z typem tablicowym.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Wsk_Ograniczone is
6    type Tablica_2_Na_2 is array (1 .. 2, 1 .. 2) of Float;
7    type Wsk_Integer is access Integer;
8    type Wsk_Float is access Float;
9    type Wsk_Wielka_Litera is access Character range 'A' .. 'Z';
10   type Wsk_Tablica_2_Na_2 is access Tablica_2_Na_2;
11   Wsk_Tab : Wsk_Tablica_2_Na_2 :=
12                   new Tablica_2_Na_2'((1..2 => 0.0), (1..2 => 0.0));
13   Wsk_I : Wsk_Integer := new Integer'(0);
14   Wsk_F : Wsk_Float := new Float'(0.0);
15   Wsk_WI : Wsk_Wielka_Litera := new Character'('A');
16
17   procedure Wypisz_Tablica (Wsk : in Wsk_Tablica_2_Na_2 ) is
18   begin
19     for I in 1..2 loop
20       New_Line;
21       for J in 1..2 loop
22         Put (Wsk (I,J));
23       end loop;
24     end loop;
25   end Wypisz_Tablica;
26
27   procedure Czytaj_Tablica (Wsk : in Wsk_Tablica_2_Na_2) is
28   begin
29     for I in 1..2 loop
30       New_Line;
31       for J in 1..2 loop
32         Get (Wsk (I,J));
33       end loop;
34     end loop;
35   end Czytaj_Tablica;
36
37 begin
38   Put ("Wypisujemy poczatkowe wartosci danych wskazywanych"); New_Line;
39   Put (Wsk_I.all); New_Line;
40   Put (Wsk_F.all); New_Line;
41   Put (Wsk_WI.all);
42   Wypisz_Tablica (Wsk_Tab);
43   Wsk_I.all := 1;
44   Wsk_F.all := 1.0;
45   Wsk_WI.all := 'B'; New_Line;
46   Put ("Podaj nowe elementy tablicy 2 na 2"); New_Line;
47   Czytaj_Tablica (Wsk_Tab); New_Line (2);
48   Put ("Wypisujemy nowe wartosci danych wskazywanych"); New_Line;
49   Put (Wsk_I.all); New_Line;
50   Put (Wsk_F.all); New_Line;
```

```
51    Put (Wsk_Wl.all);
52    Wypisz_Tablica (Wsk_Tab);
53 end Wsk_Ograniczone;
```

8.1.1 Typy wskaźnikowe ogólne

W poprzednim punkcie omówiliśmy podstawy stosowania typów wskaźnikowych ograniczonych i niektóre metody wykonywania operacji na danych tworzonych przez przydzielanie im odpowiednich obszarów pamięci, czyli danych tworzonych przez alokatory. *Typy wskaźnikowe ogólne* umożliwiają pośredni dostęp do zadeklarowanych obiektów oraz do danych tworzonych przez alokatory. Typ wskaźnikowy ogólny możemy zadeklarować w następujący sposób:

```
type Wsk_Integer is access all Integer;
```

Dzięki takiej deklaracji możemy zmiennej typu Wsk_Integer przypisać adres dowolnej zmiennej typu Integer, pod warunkiem, że zmienna ta deklarowana jest jako aliased. Piszemy więc

```
Wsk_I : Wsk_Integer;
I : aliased Integer;
..
Wsk_I := I'Access;
```

Zmiennej tej możemy nadawać wartość i przypisywać jej wartość innej zmiennej typu całkowitego przez pośrednie odwołanie za pomocą zmiennej wskaźnikowej Wsk_I. Poniższy program ilustruje takie operacje, przy czym mamy w nim również odwołania do tablicy, której składowymi są liczby typu Float.

8

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Wsk_Ogolne is
6     type Wsk_Integer is access all Integer;
7     type Wsk_Float is access all Float;
8     Wsk_I : Wsk_Integer;
9     I : aliased Integer := 100;
10    Wsk_F : Wsk_Float;
11    F : aliased Float := -1.5;
12    K : Integer := 200;
13    G : Float := 1.5;
14    Wsk_Tablica : array (1 .. 2, 1 .. 2) of aliased Float := ((1.0, 0.0),
15                                                             (0.0, 1.0));
16    procedure Wypisz_Element (I, J : Positive;
17                              X : Wsk_Float;
18                              Nazwa : String ) is
19    begin
20       Put (Nazwa); Put ("("); Put (I, 2); Put (",");
21       Put (J, 2); Put (")␣=␣");
22       Put (X.all, 3, 5, 0);
23    end Wypisz_Element;
24
```

```
25 begin
26   Wsk_I := I'access;
27   Wsk_F := F'access;
28   Put ("Oto wartosci poczatkowe zmiennych Wsk_I, Wsk_F, K i G: ");
29   New_Line;
30   Put (Wsk_I.all, 3); Put ("  ");
31   Put (Wsk_F.all, 2, 3, 0); Put ("  ");
32   Put (K, 3); Put ("  "); Put (G, 2, 3, 0);
33   K := Wsk_I.all;
34   G := Wsk_F.all;
35   New_Line;
36   Put ("Oto nowe wartosci zmiennych K i G: "); New_Line;
37   Put (K, 3); Put ("  "); Put (G, 2, 3, 0); New_Line;
38   Put ("Oto tablica: ");
39   for Indeks_Pierwszy in Wsk_Tablica'range(1) loop
40     New_Line;
41     for Indeks_Drugi in Wsk_Tablica'range(2) loop
42       Wypisz_Element (Indeks_Pierwszy, Indeks_Drugi,
43         Wsk_Tablica(Indeks_Pierwszy, Indeks_Drugi)'access, "  ");
44     end loop;
45   end loop;
46 end Wsk_Ogolne;
```

Zwracamy uwagę Czytelnika na to w jaki sposób została zmieniona deklaracja procedury Wypisz_Element, oraz w jaki sposób odwołujemy się do elementów tablicy przy jej wypisywaniu.

8

Jeżeli chcemy uniemożliwić nadawanie nowych wartości danym wskazywanym, to w deklaracji typu wskaźnikowego słowo all zastępujemy słowem constant. W celu ilustracji tej własności podajemy krótki program:

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Integer_Text_Io; use Ada.Integer_Text_Io;
3  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
4
5  procedure Wsk_Ogolne_RO is
6    type Wsk_Integer_RO is access constant Integer;
7    type Wsk_Float_RO is access constant Float;
8    Wsk_I_RO : Wsk_Integer_RO;
9    I : aliased Integer := 100;
10   Wsk_F_RO : Wsk_Float_RO;
11   F : aliased Float := -1.5;
12 begin
13   Wsk_I_RO := I'access;
14   Wsk_F_RO := F'access;
15   Put ("Oto wartosci zmiennych Wsk_I_RO, Wsk_F_RO");
16   New_Line;
17   Put (Wsk_I_RO.all, 3);
18   Put ("  ");
19   Put (Wsk_F_RO.all, 2, 3, 0);
20 -- Wsk_I_RO.all := 200; Nielegalne
21 -- Wsk_F_RO.all := -2.0; Nielegalne
22 end Wsk_Ogolne_RO;
```

Przy konstruowaniu oprogramowania dążymy zwykle do napisania możliwie uniwersalnych podprogramów, które mogą być wykorzystywane w różnych sytuacjach. Przykładem może być numeryczne obliczanie całki

$$\int_A^B f(x)dx,$$

gdzie $x \mapsto f(x) \in \Re, x \in [A, B] \subset \Re$ jest funkcją ciągłą. Jeżeli napiszemy funkcję realizującą algorytm obliczania całki np. metodą Simpsona (Wirth 1978), to chcielibyśmy zapewne, aby parametrem formalnym funkcji obliczającej całkę była funkcja podcałkowa. Przy wywołaniu funkcji realizujacej całkowanie, parametrem aktualnym, może być wtedy funkcja z której całkę chcemy wyznaczyć. W Adzie 95 mechanizmem umożliwiajacym używanie podprogramów jako parametrów innych podprogramów są wskaźniki ogólne. Dokładniej, wskaźniki takie mogą wskazywać na podprogramy. Możemy napisać:

```
type Funkcja_Rzeczywista is access function (X : Float) return Float;
Funkcja_Testowa : Funkcja_Rzeczywista;
Argument, Wartosc : Float;
```

Zmienna Funkcja_Testowa wskazuje na funkcje rzeczywiste takie jak, Sin, Cos, Sqrt czy Tan, których nagłówki są zgodne z podaną deklaracją typu wskaźnikowego np.

```
function Sat (X : Float) return Float;
```

Możemy teraz wykonać instrukcje:

```
Funkcja_Testowa := Sat'Access;
Wartosc := Funkcja_Testowa (Argument);
```

Ostatnia instrukcja podstawienia jest skróconą formą zapisu

```
Wartosc := Funkcja_Testowa.all (Argument);,
```

przy czym forma pełna jest konieczna, gdy podprogram nie ma parametrów.

Kończymy ten rozdział programem realizujacym obliczanie całki oznaczonej metodą Simpsona, przy czym funkcja podcałkowa jest parametrem wejsciowym funkcji Simpson obliczajacej sumę całkową z zadaną dokładnoscią.

```
1  with Ada.Text_Io; use Ada.Text_Io;
2  with Ada.Float_Text_Io; use Ada.Float_Text_Io;
3  with Ada.Numerics.Elementary_Functions;
4                                   use Ada.Numerics.Elementary_Functions;
5
6  procedure Metoda_Simpsona is
7
8     type Funkcja_Podcalkowa is access function
9           (X : Float) return Float;
10    T0, T1, Epsilon, Calka : Float;
11    Funkcja_Testowa : Funkcja_Podcalkowa;
12
13    function Simpson (A, B, Eps : in Float;
14                      Fun_Podcalkowa : in Funkcja_Podcalkowa )
```

8

```
15      return Float is
16      -- A < B - konce przedzialu calkowania, Eps - zadana dokladnosc
17      I, N : Integer;
18      S, Ss, S1, S2, S4, H : Float;
19    begin
20      N := 2;
21      H := (B-A)/2.0;
22      S1 := H*(Fun_Podcalkowa(A) + Fun_Podcalkowa(B));
23      S2 := 0.0;
24      S4 := 4.0*H*Fun_Podcalkowa(A+H);
25      S := S1 + S2 + S4;
26      loop
27        Ss := S;
28        N := 2*N;
29        H := H/2.0;
30        S1 := 0.5*S1;
31        S2 := 0.5*S2 + 0.25*S4;
32        S4 := 0.0;
33        I := 1;
34        loop
35          S4 := S4 + Fun_Podcalkowa(A + Float(I)*H);
36          I := I + 2;
37          exit when I > N;
38        end loop;
39        S4 := 4.0*H*S4;
40        S := S1 + S2 + S4;
41        exit when (abs(S - Ss) < Eps*abs(S));
42      end loop;
43      return S/3.0;
44    end Simpson;
45  begin
46    Funkcja_Testowa := Cos'access;
47    Put ("Podaj lewy koniec przedzialu calkowania ");
48    Get (T0);
49    Put ("Podaj prawy koniec przedzialu calkowania ");
50    Get (T1);
51    Put ("Przedzial calkowania = ");
52    Put (T0, 3, 5, 0); Put (" .."); Put (T1, 3, 5, 0);
53    New_Line;
54    Put ("Podaj dokladnosc "); Get (Epsilon);
55    Put ("Dokladnosc = "); Put (Epsilon, 3, 5, 0); New_Line;
56    Calka := Simpson (T0, T1, Epsilon, Funkcja_Testowa);
57    Put ("Wartosc calki = "); Put(Calka, 3, 5, 0);
58  end Metoda_Simpsona;
```

Program ten wyprodukował następujace wyniki:

```
Przedzial calkowania = -1.00000 .. 1.00000
Dokladnosc = 0.01000
Wartosc calki = 1.68354
```

8.2 Zwalnianie pamięci

Czytelnik zaniepokoił się zapewne co się dzieje z pamięcią zaalokowaną przy pomocy operatora new. Przecież nie jest trudno zażądać tyle pamięci, że system nie będzie w stanie jej dostarczyć i co wtedy? Wtedy można by oddać część wcześniej zabranej pamięci systemowi do puli, z której może udostępniać ją kolejnym żądaniom.

Filozofia oddawania pamięci w różnych językach programowania jest różna. Zwykle, tak jak to jest w C, C++, Pascalu i wielu, wielu innych językach trzeba jawnie zwolnić pamięć przy pomocy operatora odwrotnego do new. W innych językach – np. w Javie (Eckel 2001) pamięć jest zwalniana automatycznie przez proces systemowy zwany „*odśmiecaczem*" (ang. *garbage collector*).

Wydawać się może, że to drugie rozwiązanie jest lepsze, ponieważ zwalnia programistę od pamietania o konieczności zwalniania pamięci. Jednakże ma też bardzo poważną wadę polegającą na tym, że proces „odśmiecacza" włączając się w pewnej chwili, na którą programista nie ma żadnego wpływu, blokuje działanie innych zadań w systemie co może doprowadzić do niewłaściwego działania całego systemu (proszę sobie wyobrazić, że samolot zaprzestał wysuwania podwozia przy lądowaniu tylko „na chwilę") (Kopetz 1998, www.rtj.org 2001).

W różnych implementacjach Ady może, ale nie musi się znajdować proces „odśmiecacza". Jeżeli taki proces istnieje, to problem zwalniania pamięci jest rozwiązany. Jeżeli jednak taki proces nie istnieje, to trzeba dać programiście możliwość zwolnienia pamięci. W Adzie nie ma operatora odwrotnego do new, ale istnieje prosta metoda, by go sobie dopisać. Załóżmy, że procedura ta będzie się nazywać wzorem C++ ⇒ delete.

Język Ada umożliwia stosowanie tzw. ogólnych jednostek programowych (zob. rozdział 13), czyli podprogramów i pakietów (rozdział 9), których parametrami nie są zmienne ale typy. Posłużmy sie przykładem typu Wezel i wskaźnika Wskaznik zdefiniowanego na stronie 178 przy okazji omawiania struktur drzewiastych. Wtedy deklaracja:

```
1  with Ada.Unchcecked_Deallocation; -- specyfikacja biblioteki
2                                    -- NIE MA odpowiednika "use"
3  ...
4  ...
5
6  procedure delete is
7      new Ada.Unchecked_Deallocation (Wezel, Wskaznik);
8
9  ...
10 ...
11
12 a : Wskaznik := new (Wezel);
13 -- operacje na zmiennej a
14 delete (a);
```

Konkretyzacja procedury Ada.Unchecked_Deallocation wymaga podania typu, który jest zaalokowany i odpowiedniego wskaźnika, natomiast użycie wymaga podania jako argumentu procedury delete tylko wskaźnika.

Mamy nadzieję, że jest oczywiste, że procedurę delete można dowolnie przeciążać i nic nie stoi na przeszkodzie, by przy definiowaniu każdego typu wskaźnikowego dodawać odpowiednią konkretyzację procedury Ada.Unchecked_Deallocation. Więcej informacji ma temat konkretyzacji, w szczególności na temat konkretyzacji procedur ogólnych można znaleźć w rozdziale 13.

Rozdział 9

Pakiety

Wszystkie dotychczasowe rozważania dotyczyły pisania jednego programu. Nietrudno jednak zauważyć, że na początku każdego programu ilustrującego omawiane konstrukcje i własności Ady, znajdują się informacje postaci:

```
with Nazwa_Pakietu; use Nazwa_Pakietu;
```

Dzięki tym informacjom w programie można wykorzystywać obiekty określone w pakiecie Nazwa_Pakietu. Najczęściej korzystaliśmy z pakietów zawierających określenia i implementacje procedur i funkcji służących do pobierania danych z domyślnego urządzeniu wejściowego, jakim jest klawiatura alfanumeryczna, oraz do wypisywania danych na domyślnym urządzeniu wyjściowym, którym jest okno tekstowe na ekranie monitora. Dzięki temu, że stosowaliśmy w naszych programach obiekty udostępniane przez pakiety, nie musieliśmy pisać własnych procedur służących np. do wczytywania danych typu Float, albo wypisujących odpowiednie napisy na ekranie. Możliwość taka wynika z tego, że w Adzie możemy tworzyć oddzielne *jednostki programowe* nazywane *pakietami*, które zawierają wszystkie potrzebne deklaracje i szczegóły implementacyjne obiektów, które mają być udostępniane innym programom i pakietom. Charakterystyczne jest przy tym to, że w pakietach występują oddzielnie *część publiczna* i *część implementacyjna* (nazywana w Adzie *treścią pakietu*, ang. *body*). W części publicznej umieszcza się definicje typów, deklaracje stałych i zmiennych oraz nagłówki procedur i funkcji, które mają być dostępne w innych jednostkach programowych, natomiast część implementacyjna zawiera wszystkie deklaracje związane z implementacją algorytmów realizowanych przez publiczne procedury i funkcje, oczywiście może zawierać także deklaracje typów, konkretyzacje pakietów oraz podprogramy potrzebne do realizacji implementacji. Część implementacyjna nie jest dostępna na zewnątrz pakietu i bardzo często nie interesuje nas jaki algorytm realizuje importowany do naszego podprogram, a ważne jest, że możemy ten program zastosować. Oczywiście, polegamy tu na umiejętnościach programistów, którzy napisali część implementacyjną pakietu, z którego korzystamy i dopiero, gdy otrzymujemy podejrzane wyniki powinniśmy interesować się szczegółami implementacyjnymi.

Definicja

Pakietem nazywamy grupę logicznie powiązanych elementów, które mogą być typami, podtypami, obiektami tych typów i podtypów, oraz mogą być podprogramami z parametrami tych typów i podtypów. Co więcej pakiet może także zawierać inne pakiety. Niezwykle istotne jest by pamiętać, że pakiety mogą korzystać z elementów które pochodzą z innych pakietów.

Składnia części publicznej pakietu w notacji EBNF

```
package_declaration ::= package_specification;
package_specification ::=
    package defining_program_unit_name is
        {basic_declarative_item}
    end [defining_program_unit_name];
defining_program_unit_name ::= [parent_unit_name.]identifier
basic_declarative_item ::=
    type_declaration |
    subtype_declaration |
    object_declaration |
    subprogram_declaration |
    generic_instatiation
```

Składnia części implementacyjnej pakietu w notacji EBNF

```
package_body ::=
    package body defining_program_unit_name is
        [declarative_part]
    end [defining_program_unit_name];
declarative_part ::=
    {
        basic_declarative_item |
        subprogram_body
    }
```


Weźmy teraz pod uwagę prostą obsługę stosu reprezentowanego przez tablicę i zmienną indeksującą element znajdujący się na wierzchu stosu. Obsługę stosu realizuje się przy pomocy dwóch podprogramów Push i Pop, które odpowiednio dodają i usuwają element ze stosu. W celu udostępnienia tych procedur innym jednostkom programowym możemy stworzyć pakiet Stos (Barnes 1998).

```
1  package Stos is
2    procedure Push (X : Integer );
3    function Pop (X : Integer )return Integer;
4  end Stos;
5
6  package body Stos is
7    Max : constant Integer := 100;
8    St : array (1 .. Max) of Integer;
9    Wierzcholek : Integer range 0..Max;
10
11   procedure Push (X : Integer ) is
12   begin
13     Wierzcholek := Wierzcholek + 1;
```

```
14      St (Wierzcholek) := X;
15    end Push;
16
17    function Pop (X : Integer ) return Integer is
18    begin
19      Wierzcholek := Wierzcholek - 1;
20      return St(Wierzcholek + 1);
21    end Pop;
22
23  begin
24    Wierzcholek := 0;
25  end Stos;
```

Pakiet ten składa się z dwóch części: publicznej, albo definicyjnej, która znajduje się w pliku «*stos.ads*» i części implementacyjnej, która jest w pliku «*stos.adb*». Pliki te kompiluje się oddzielnie, przy czym pierwszy kompilowany musi być plik z częścią definicyjną[1]. Część definicyjna zaczyna się słowem kluczowym package, po którym mamy nazwę pakietu i słowo is. Potem podaje się wszystkie deklaracje typów stałych i zmiennych oraz nagłówki podprogramów, które pakiet udostępnia i ta część kończy się słowem end z nazwą pakietu. Część implementacyjna zaczyna się od słów package body, potem mamy deklaracje potrzebne w tej części i pełne deklaracje podprogramów. Część ta może mieć też *część inicjującą*, która, jeżeli występuje, jest ciągiem instrukcji po słowie kluczowym begin. Część implementacyjna, podobnie jak definicyjna zakończona jest słowem end i nazwą pakietu. Zauważmy, że w podanym przykładzie wszystkie deklaracje potrzebne do implementacji podprogramów obsługi stosu znajdują się w części implementacyjnej i nie są publiczne, natomiast część inicjująca nie była potrzebna, bo zmienną Wierzcholek mogliśmy zainicjować przy jej deklaracji.

9.1 Logiczne odmiany pakietów

Pakiety definiujące zawierają określenia stałe i typy używane w kilku programach, albo używane przez różnych programistów opracowujących różne części dużego programu. Pakiety takie **nie mają** części implementacyjnej!

Pakiety usługodawcze zawierające stałe, typy, podtypy, zmienne i podprogramy konieczne do obsługi pewnych usług. Przykładami takich pakietów są Ada.Text_IO, czy Ada.Numerics.Elementary_Functions.

Pakiety do abstrakcji danych służące do określenia dodatkowych operacji, które mogą być wykonywane na danych typu określonego przez programistę.

9.2 Typy prywatne

Jak wspomniano powyżej podstawową funkcją samodzielnych jednostek kompilacji (w Adzie i Javie – pakietów, w Turbo Pascalu – jednostek, w Moduli-2 –

[1] Zwykle przy kompilacji części implementacyjnej, kompilacja wcześniej nieskompilowanych pakietów definicyjnych następuje automatycznie.

modułów[2]) jest odseparowanie od siebie tych jednostek w taki sposób, by mogły one korzystać wzajemnie ze swoich usług w sposób inny niż przewidziany przez autora danego pakietu.

Jeżeli pewien pakiet opisuje operacje plikowe (zob. rozdział 9.7), to definiuje on m.in. operację otwierania i zamykania pliku, czytania i pisania do niego. Ukrywa on jednak informacje związane z buforowaniem zapisów, opisami plików itp. Informacje te znajdują się w pewnej strukturze, która musi być znana kompilatorowi (ponieważ konieczna jest właściwa alokacja pamięci), ale która jest niedostępna programiście korzystającemu z danego pakietu (programista implementujący ten pakiet ma oczywiście dostęp do wszystkich jego struktur). Np.

```
package File_System is
  type File_Handle is private;
  type File_Mode is (Write, Read);

  function Open( FileName : string; typ : File_Mode ) return File_Handle;
  ...
private
  type File_Handle is record
    ...
    Bufor : array(0..1023) of byte;
    ...
  end record;
end File_System;
```

Niezależnie od postaci rekordu File_Handle, która może być zmieniana w trakcie rozwoju oprogramowania wszystkie jednostki programowe korzystające z funkcji File_System.Open będą działać poprawnie. Można tworzyć i operować na zmiennych typu File_Handle, ale niemożliwy jest dostęp do np. zmiennej Bufor. Nawet jeżeli definicja zmiennej miałaby postać:

```
type File_Handle is new Natural;
```

to niemożliwe byłoby – z całą pewnością omyłkowe – dodanie do siebie zmiennych tego typu, co byłoby możliwe, gdyby typ File_Handle nie był typem prywatnym.

Jedyne operacje, które można przeprowadzić bezpiecznie na zmiennych typu File_Handle to operacje postawienia i porównania („:=”, „=” i „/=”). Oczywiście mowa tutaj o operatorach domyślnych porównujących (kopiujących) bajt po bajcie dwie zmienne typu File_Handle:

```
a, b : File_Handle;
...
a := b;
...
if a = b then
  ...
end if;
```

[2] W C++ (Stroustrup 1991) podobną rolę pełnią klasy, ale opis klasy nie musi się znajdować w jednym pliku i niekoniecznie wymaga oddzielnej kompilacji.

9.3 Typy ograniczone

Bywa jednak i tak, że należy uniemożliwić programiście korzystającemu z danego modułu używania nawet domyślnych operacji porównania i kopiowania, skazując go tylko na operacje zdefiniowane w tym module. Typowym przykładem, w którym konieczne jest takie podejście jest przypadek, w którym zmienna typu byłaby np. listą[3]. Dwie listy sa takie same jeśli mają takie same elementy wyłączywszy same wskaźniki, które są oczywiście różne. W przypadku typu File_Handle wydaje sie, że porównywanie zawartości bufora w teście na równość również nie ma większego sensu.

W takim przypadku należałoby zdefiniować typ File_Handle jako:

```
type File_Handle is limited private;
```

Oczywiście parametry procedur typu File_Handle, muszą być w takim przypadku przekazywane wyłącznie w trybie out lub in out.

Dalsze informacje na ten temat znajdzie czytelnik w rozdziałach 12, (w szczególności 12.7) i 13.

9.4 Pakiety zagnieżdżone

Tak jak procedura może zawierać procedury lokalne, tak pakiet może zawierać podpakiety zależne od pakietu głównego zwane pakietami zagnieżdżonymi (*ang. nested package*). Dobrym przykładem takiego pakietu jest pakiet Ada.Text_IO.Integer_IO zagnieżdżony w pakiecie Ada.Text_IO, (zresztą z wieloma sobie podobnymi).

Najczęściej takimi pakietami są pakiety ogólne (rozdział 13), choć takie ograniczenie nie wynika z języka (zobacz też program na stronie 196).

9.5 Pakiety potomne

Często występuje taka sytuacja, że dany pakiet składa się na tak złożony fragment całości oprogramowania, że nie może go opracować jedna osoba, albo też pakiet ten byłby za duży (dobry obyczaj nakazuje, by pojedynczy pakiet nie przekraczał kilkuset linii kodu) i należy go rozbić na logicznie odseparowane części.

Sposobem by to zrobić w języku Ada są tzw. pakiety potomne (ang. *Child Packages*)

Jak wspomnieliśmy wcześniej, Ada posiada hierarchiczna strukturę pakietów pomagającą w ich organizacji. Wszystkie pakiety standardowe Ady wywodzą się z trzech pakietów: Ada, Interfaces i System. Pakiety te nazywamy macierzystymi. Pakiet Ada jest pakietem macierzystym większości innych standardowych pakietów bibliotecznych Ady. Zasoby pozwalające na łączenie programów pisanych w Adzie z bibliotekami pisanymi w innych językach można znaleźć w

[3]Choć naprawdę w tym przypadku trudno byłoby to uzasadnić.

pakiecie Interfaces i jego pakietach potomnych. Pakiet System i jego pakiety potomne dostarczają definicji charakterystyk otoczenia systemowego w jakim mają działać programy Ady. Oczywiście pakiety potomne mogą mieć swoje pakiety potomne, a te znów swoje itd.

Pakiet standardowy Elementary_Functions jest pakietem potomnym pakietu Numerics, który jest z kolei pakietem potomnym pakietu Ada. Strukturę tę zapisujemy następująco: Ada.Numerics.Elementary_Functions.

Trzeba pamiętać, że pakiet potomny automatycznie ma dostęp do wszystkich obiektów pakietów macierzystych, także tych prywatnych. Stanowi to ekwiwalent znanych np. z języka C++ modyfikatorów protected[4]. Rozważmy następujące deklaracje

Definicja pakietu macierzystego

```
1  package PM is
2
3      procedure P1;
4      procedure P2;
5
6      package Nested is
7          procedure N;
8      end Nested;
9
10 private
11     procedure Priv;
12 end PM;
```

Definicja pakietu potomnego (zwykle stosuje się odpowiednie nazewnictwo plików – często nazwy pakietów oddzielone są znakiem myślnika – w tym przypadku pm−pp.ads)

```
1  package PM.PP is
2
3      procedure P3;
4
5  end PM.PP;
```

Implementacja pakietu potomnego

```
1  -- Brak listy importowej dotyczącej pakietu PM.
2  package body PM.PP is
3
4      procedure P3 is
5      begin
6          P1; -- Procedury pakietu są dostepne bezpośredni
7          P2;
8          Priv; -- Także procedura prywatna!
9          Nested.N; -- Dostęp do pakietu zagnieżdżonego
10                   -- wymaga kwalifikacji
11     end P3;
```

[4] Jak widać obiekt (typ, podprogram) **prywatny** w języku Ada odpowiada obiektowi **chronionemu** w języku C++, natomiast odpowiednikiem obiektu **prywatnego** w rozumieniu języka C++, są w Adzie obiekty lokalne zadeklarowane w części implementacyjnej modułu, do których, w oparciu o ogólne zasady widoczności, mają dostęp wyłącznie inne obiekty tego modułu.

```
12
13 end PM.PP;
```

9.6 Operacje wejścia/wyjścia

W rozdziale tym ograniczymy się do operacji wejściowo-wyjściowych na danych tekstowych. Dodatkowo przyjmujem, że mamy dwa standardowe pliki – jeden do danych wejściowych i drugi do danych wyjściowych. Przyjmujemy też, że dane wejściowe wprowadzane sa przy pomocy klawiatury (albo ze strumienia wejściwego), a dane wyjściowe wyprowadzane są na ekran (albo do strumienia wyjściowego). Pakietem, który zawiera odpowiednie dla naszych potrzeb operacje wejścia/wyjścia jest pakiet standardowy Ada.Text_IO i jego pakiety zagnieżdżone Ada.Text_IO.Integer_IO, Ada.Text_IO.Enumeration_IO i inne. W pakiecie Ada.Text_IO zdefiniowany jest m.in.

```
type Count is range 0 .. Natural'Last;
```

oraz procedury i funkcje

```
procedure New_Line (Spacing : in Count := 1);
procedure Set_Col (To : in Count);
function Col return Count;
procedure Get (Item : out Character);
procedure Get (Item : out String);
procedure Get_Line (Item : out String;
                    Last : out Natural);
procedure Put (Item : in Character);
procedure Put (Item : in String);
procedure Put_Line (Item : in String);
```

Jeżeli wywołamy w naszym programie procedurę New_Line bez parametru, to następne wypisywane dane będą się pojawiać od początku następnej linii[5] Możemy oczywiście pisać

```
New_Line (1)
```

ale nie jest to potrzebne, ponieważ w nagłówku procedury została podana wartość domyślna parametru wejściowego Spacing. Procedura Set_Col oraz funkcja Col służą do tabulacji. Położenie znaku w linii, w której wypisywane są dane, liczone jest od 1. Jeżeli w naszym programie napiszemy instrukcję Set_Col (10);, to następny wypisywany znak pojawi się w pozycji 10. Funkcja Set_Col (Col + 10) przesuwa aktualna pozycję o 10 w prawo. Procedury Put sa przeciążone i w zależności od typu aktualnego parametru wejściowego dobierane jest odpowiednie wywołanie:

```
Put ('A');
Put ("To_jest_napis");
```

Przy wypisywaniu napisów często po wypisywaniu tekstu chcemy przejść do pisania od nowej linii. Możemy napisać

[5] Procedura jest zależna od implementacji – w systemach produkcji Microsoft i podobnych koniec linii oznaczany jest przez parę znaków ASCII.cr i ASCII.lf, w systemach Unix i podobnych – przez pojedynczy znak ASCII.lf.

```
Put ("To␣jest␣napis");
New_Line;
```

ale można krócej

```
Put_Line("To␣jest␣napis")
```

Analogicznie do procedur Put, do czytania danych służą funkcje Get. Trzeba jednak pamiętać, że wersja tej procedury służąca do czytania napisów oczekuje, że przeczytanych zostanie dokładnie tyle znaków, ile „zmieści" się w tym napisie. Jeżeli jednak życzymy sobie, by zostały przeczytane wszystkie znaki do momentu pojawienia się znaku końca linii, to należy użyć procedury Get_Line, w której Item jest „wystarczająco dużym" napisem, zaś, Koniec jest ostatnim wczytanym znakiem. Wczytane znaki są dostępne jako Item (Item'First .. Koniec). Jeżeli Koniec = 0, to jest to napis pusty.

Jeżeli chcemy wykonać operację wejścia/wyjścia na danych typu Float, to możemy wykorzystać pakiet Ada.Float_Text_IO, lub skonkretyzować dla tego typu (rozdział 13) pakiet Ada.Text_IO.Float_IO: package FIO is new Ada.Text_IO.Float_IO (Float).

Jeżeli A : Float, to pisząc Put (A); otrzymamy wynik w postaci wykładniczej (tzw. naukowej), przy czym dostaniemy 7 cyfr znaczących, jedną przed znakiem dziesiętnym (tzn. kropką), i sześć po tym znaku. Do tego dochodzi znak spacji (liczba dodatnia lub zero), albo znak minus, gdy liczba jest ujemna. Wykładnik składa się z litery E, po której mamy znak +/− i 2 cyfry wykładnika. Wynik instrukcji Put (12.34) jest więc następujący:

```
␣1.234000E+01
```

Jeżeli chcemy zmienić formę wyniku, to mamy do dyspozycji trzy parametry, które odpowiednio podają: ilość znaków przed kropką dziesiętną, liczbę znaków po kropce dziesiętnej i ilość znaków w wykładniku (tzn. po literze E). Np. Put (12.34, 3, 4, 2) daje

```
␣␣1.2340E+1
```

zaś Put (12.34, 3, 4, 0) daje

```
␣12.3400
```

Zobacz też rozdział 13.4.

9.7 System plików

Dotychczas zapoznaliśmy się z czytaniem i pisaniem na konsoli, czyli czytaniem znaków (także liczb, napisów itp.) z klawiatury i ich wypisywaniem na ekranie.

Czytelnicy wiedzą już, że procedury i funkcje umożliwiające takie operacje są zgromadzone w pakiecie Ada.Text_IO. Jednakże w tym pakiecie zgromadzone są, prócz już nam znanych, także i inne bardzo użyteczne procedury. Np. zamiast wypisywać wartość na ekran możemy ją zachować w tekście (np. Put (str, Zmienna_Całkowita), możemy z tego napisu ją przeczytać (np. Get (str, Zmienna_Całkowita)).

Jednakże najbardziej interesującym, a dotąd czytelnikom nieznanym, aspektem tego pakietu jest możliwość zapisywania wyników obliczeń w plikach dyskowych, a także odzyskiwania z nich użytecznych informacji.

W języku Ada pliki (czyli abstrakcje opisujące pewien zbiór na dysku, w pamięci, potok komunikacyjny, także urządzenie) mogą mieć różny charakter w zależności od struktury wewnętrznej zawartych w niej informacji (W Turbo Pascalu, istniały typy file, file of text, file of Pewien_Typ). W języku Ada odpowiedni typ danych nazywa się zawsze tak samo File_Type, a struktura wewnętrzna pliku opisywanego przez daną typu File_Type wynika z tego, z jakiego pakietu jest on importowany.

I tak zmienna typu:

- ⇝ Text_IO.File_Type opisuje pliki tekstowe. Stosując takie pliki należy koniecznie pamiętać o tym, że czytając i zapisując do takiego pliku jednocześnie interpretujemy znaki sterujące (np. znaki ograniczające linie tekstu w systemie Unix (Character'Val (10)[6]) i w systemie DOS, Windows (Character'Val (13)& Character'Val (10)[7]) są interpretowane w taki sposób, że są zawsze zamieniane na znak (Character'Val (30)[8])

- ⇝ Direct_IO.File_Type opisuje pliki, których elementy są dowolnego typu i nie podlegają żadnej interpretacji. Pakiet Direct_IO jest pakietem ogólnym (13) i jego konkretyzacja dla typu Character, a jeszcze lepiej dla typu zdefiniowanego następująco (zobacz też rozdział 11):

  ```
  type BYTE is mod 256;
  for BYTE'Size use 8;
  ```

  jest tożsama z plikiem nie posiadającym struktury wewnętrznej.

  Jednakże trzeba pamiętać, że jego konkretyzacja dla dowolnego innego typu nadaje plikowi strukturę wewnętrzną.

- ⇝ Sequential_IO.File_Type opisuje w zasadzie takie same pliki jak Direct_IO.File_Type (jest to również pakiet ogólny), z tym, że nie istnieją dla niego zdefiniowane operacje pozycjonowania (jest to użyteczne np. w przypadku potoków).

- ⇝ Ada.Streams.Stream_IO.File_Type opisuje podstawowe operacje na strumieniach (ang. *stream*)

9.7.1 Podstawowe operacje na plikach

Korzystając z systemu plików użytkownik ma zawsze dostęp do podprogramów takich jak:

```
1  procedure Create
2    (File : in out File_Type;
3     Mode : in File_Mode := Out_File;
```

[6] ASCII.lf = *Line Feed*
[7] ASCII.lf i ASCII.cr = *Carriage return*
[8] ASCII.eol = *End of Line*

```
4         Name : in String := "";
5         Form : in String := "");
6
7     procedure Open
8       (File : in out File_Type;
9        Mode : in File_Mode;
10       Name : in String;
11       Form : in String := "");
12
13    procedure Close (File : in out File_Type);
14    procedure Delete (File : in out File_Type);
15    procedure Reset (File : in out File_Type; Mode : in File_Mode);
16    procedure Reset (File : in out File_Type);
17
18    function Mode (File : in File_Type) return File_Mode;
19    function Name (File : in File_Type) return String;
20    function Form (File : in File_Type) return String;
```

przy czym typ File_Mode jest zdefiniowany jako

```
type File_Mode is (In_File, Out_File, Append_File);
```

Uwaga!

Trzeba pamiętać, że do pliku otworzonego w trybie In_File nie można pisać, ale można z niego czytać, z pliku otworzonego w trybie Out_File nie można czytać, a tylko do niego pisać, a i pisać i czytać można tylko z pliku otworzonego w trybie Append_File. Z tym faktem związane są nierozerwalnie jego skutki, takie, że można wielokrotnie otworzyć plik w trybie In_File, a także jest to jedyny sposób w jaki można dostać się do zawartości plików z ustawionym atrybutem *Read Only*, albo *System*, albo znajdujących się w katalogu, do którego użytkownik nie ma praw do zapisu. Jeżeli plik jest otwarty w trybie Out_File, albo Append_File to użytkownik uzyskuje do niego wyłączny dostęp i nikt (także on sam) nie może tego pliku otworzyć po raz drugi (nie zamykając go wcześniej). Wyjaśnienie powodu dlaczego pliku w trybie innym niż In_File nie można stworzyć/otworzyć wielokrotnie pozostawiamy Czytelnikom.

Plik przed użyciem trzeba koniecznie otworzyć (Open) lub stworzyć (Create), następnie wykonać szereg operacji zapisu i odczytu na tym pliku i w końcu plik zamknąć poleceniem Close.

Każdy system operacyjny pozwala na jednoczesne otwarcie ograniczonej liczby plików i, nawet jeżeli jest to duża liczba, niezamykanie plików może spowodować, że program nie będzie w stanie otworzyć kolejnego ze względu na wyczerpanie się zasobów systemowych i, chociaż po zakończeniu programu użytkownika systemy operacyjne starają się „posprzątać” po użytkowniku m.in. zamykając pootwierane pliki, dobry obyczaj nakazuje jednak użytkownikowi zamykanie plików natychmiast po tym, gdy zaniknie potrzeba do nich dostępu.

Na plik dyskowy należy patrzeć tak, jak na taśmę magnetofonu – żeby odczytać pewną informację należy plik przewinąć, co czasami jest operacją kosztowną czasowo (stąd używanie plików sekwencyjnych). Z dostępem do pliku związany jest wskaźnik wskazujący w jakim miejscu zostanie zapoczątkowana kolejna operacja zapisu lub odczytu. Przeczytanie (i zapisanie) dowolnej informacji powo-

duje przesunięcie tego wskaźnika o wielkość tej informacji (tak jak odsłuchanie (ale i nagranie) piosenki z magnetofonu powoduje przesunięcie taśmy.

Oczywiście w przypadku pakietów pozwalających na dostęp swobodny do dysku (np. Direct_IO) możliwe jest przesuwanie tego wskaźnika:

```
procedure Set_Index (File : in File_Type; To : in Positive_Count);
function Index (File : in File_Type) return Positive_Count;
```

Dodatkowe funkcje związane z obsługą plików znajdzie Czytelnik i Czytelniczka w opisie bibliotek systemowych, do lektury których zachęcamy. Zwracamy też uwagę na to, że konkretyzacja np. pakietu ogólnego Ada.Text_IO.Integer_IO powoduje także automatyczny dostęp do formatowanych operacji wejścia wyjścia także na plikach dyskowych.

Nie można jednak nie wspomnieć o błędach związanych z obsługą plików.

Najbardziej oczywista jest funkcja:

```
function End_Of_File (File : in File_Type)return Boolean;
```

pozwalająca sprawdzić czy aktualny wskaźnik pliku znajduje się na jego końcu. Nie zalecamy jednak stosowania tej funkcji, natomiast zalecamy korzystanie z obsługi wyjątków (rozdział 10), co pozwala na tworzenie znacznie bardziej czytelnego i efektywnego kodu.

Wyjątki związane z obsługa plików zgromadzone sa w pakiecie IO_Exceptions i odsyłamy Czytelnika do znajdującego się tam opisu (zresztą angielskie nazwy tych wyjątków całkiem nieźle opisują ich znaczenie) tu ograniczając się do ich wymienienia: Status_Error, Mode_Error, Name_Error, Use_Error, Device_Error, End_Error, Data_Error.

Pamiętajmy, że bardziej czytelny (i bardziej efektywny) jest program:

```
1 loop
2     Cos;
3 end loop
4 exception
5     when End_Error =>
6         Close (Plik);
```

niż równoważny mu

```
1 while not End_Of_File (Plik) loop
2       Cos;
3 end loop
4 Close (Plik);
```

9.7.2 Strumienie

Strumieniem nazywamy taką „strugę” danych, która nie ma jednorodnego charakteru wewnętrznego. Strumień może być wykorzystywany do zapisu na dysku, ale może być też wykorzystany do innych zadań o podobnym charakterze – np przesyłania danych poprzez sieć komputerową np. za pośrednictwem gniazd

TCP[9]. Strumień jest implementowany w Adzie w taki sposób, że każdej strukturze logicznej programu odpowiada specjalnie zadeklarowana tablica otwarta – Ada.Streams.Stream_Element_Array. Użytkownik może zdefiniować odpowiednie podtypy na bazie typu Ada.Streams.Stream_Element_Array o wielkości równej odpowiedniej strukturze programu użytkownika a następnie za pomocą konkretyzacji funkcji Ada.Unchecked_Conversion zamieniać odpowiednie struktury.

Poniżej przedstawiamy przykładowy program wczytujący z pliku binarnego tablicę składającą się z elementów typu RGB, przy czym na początku pliku znajduje się liczba informująca o wielkości tej tablicy. Zwracamy uwagę, że przy takim podejściu użytkownik odpowiada tylko za zgodność ze strukturą pliku i raczej nie może się pomylić w jego interpretacji. Zobacz też pakiet ogólny Print na stronie w rozdziale 13.4 na stronie 296. Ponadto warto pamietać, że w praktyce zwykle korzysta się nie wprost ze struktur danych, tak jak to zademonstrowano w poniższym przykładzie, ale raczej ze wskaźników do tych struktur.

```
1  with Text_IO;
2  with Ada.Streams; use Ada.Streams;
3  with Ada.Streams.Stream_IO; use Ada.Streams.Stream_IO;
4  with Ada.Unchecked_Conversion, Print;
5
6
7  procedure Streams_Test is
8
9      type Byte is new Integer range 0 .. 255;
10     for Byte'Size use 8;
11
12     package BytePrint is new Print (Byte); use BytePrint;
13     package SEOPrint is new Print (Stream_Element_Offset); use SEOPrint;
14
15     type RGB is
16         record
17             R, G, B : Byte;
18         end record;
19
20     subtype Wlk_SA is Stream_Element_Array (1 .. Integer'Size / 8);
21     subtype RGB_SA is Stream_Element_Array (1 .. RGB'Size / 8);
22
23
24     function To_Stream is new Ada.Unchecked_Conversion (Integer, Wlk_SA);
25     function To_Stream is new Ada.Unchecked_Conversion (RGB, RGB_SA);
26     function To_Int is new Ada.Unchecked_Conversion (Wlk_SA, Integer);
27     function To_RGB is new Ada.Unchecked_Conversion (RGB_SA, RGB);
28
29     Wielkosc : Integer := 7;
30     Wlk : Wlk_SA;
31
32     Last : Stream_Element_Offset;
33
34     Plik : File_Type;
35 begin
36     Create (Plik, Out_File, "Plik.testowy");
37     Write (Plik, To_Stream (Wielkosc));
```

[9] Jest to dziś najbardziej powszechny sposób komunikacji w sieciach komputerowych.

```
38    for idx in 1 .. Wielkosc loop
39       declare
40          kolor : RGB := (R => Byte (idx),
41                          G => Byte (2 * idx),
42                          B => Byte (3 * idx));
43       begin
44          Write (Plik, To_Stream (kolor));
45       end;
46    end loop;
47    Close (Plik);
48    -- A teraz przeczytamy co zapisaliśmy
49    Open (Plik, In_File, "Plik.testowy");
50    Read (Plik, Wlk, Last);
51    Text_IO.Put_Line ("Wartosc ""Last"" = %d" & Last);
52    for idx in 1 .. To_Int (Wlk) loop
53       declare
54          kolor_SA : RGB_SA;
55       begin
56          Read (Plik, kolor_SA, Last);
57          Text_IO.Put_Line ("Last = %d, "
58                            & "kolor.R = %d, "
59                            & "kolor.G = %d, "
60                            & "kolor.B = %d"
61                            & Last
62                            & To_RGB (kolor_SA).R
63                            & To_RGB (kolor_SA).G
64                            & To_RGB (kolor_SA).B
65                            );
66       end;
67    end loop;
68    Close (Plik);
69 end Streams_Test;
```

Program wyprodukował wynik:

```
1 Wartosc "Last" = 4
2 Last = 3, kolor.R = 1, kolor.G = 2, kolor.B = 3
3 Last = 3, kolor.R = 2, kolor.G = 4, kolor.B = 6
4 Last = 3, kolor.R = 3, kolor.G = 6, kolor.B = 9
5 Last = 3, kolor.R = 4, kolor.G = 8, kolor.B = 12
6 Last = 3, kolor.R = 5, kolor.G = 10, kolor.B = 15
7 Last = 3, kolor.R = 6, kolor.G = 12, kolor.B = 18
8 Last = 3, kolor.R = 7, kolor.G = 14, kolor.B = 21
```

Część II

Zaawansowane możliwości Ady

Rozdział 10

Wyjątki

Nie jest żadną tajemnicą, że programy, nawet te dobrze przetestowane (co, niestety, jest dziś coraz rzadszym obyczajem), mają błędy. Niestety, większość języków programowania jest tak zaprojektowana, żeby można było błędy ignorować. Podejście takie można uzasadnić (ale nie usprawiedliwić!) tym, że błędy w programach, które zostały oddane do użytkowania, zdarzają się stosunkowo rzadko, a nakład pracy związany z programową obsługą takich błędów (i zwolnienie działania programu) jest znaczny. Ponadto, zaprogramowanie takiej obsługi znacząco zmniejsza też czytelność programu. Szczególne „zasługi" w ułatwianiu programiście pomijania sytuacji wyjątkowych ma język C (ale już nie C++). Dla tych, którzy znają ten język, a nie zgadzają się z powyższym stwierdzeniem proponujemy by spojrzeli w napisane przez siebie programy i sprawdzili jak często sprawdzają poprawność wykonania np. funkcji printf. A może nigdy?

Ada jest pierwszym językiem, w którym zaimplementowano obsługę wyjątków, a inne, bardziej dziś popularne języki programowania, takie jak C++, czy Java zapożyczyły tę ideę właśnie od Ady (Eckel 2001, p. 391)

Zważywszy na to, że Ada jest pomyślana jako język stosowany w konstrukcji oprogramowania urządzeń o najwyższej odpowiedzialności takich jak reaktory, systemy sterowania lotem czy systemy telekomunikacyjne, oraz na to, że założenie o bezbłędności oprogramowania jest mrzonką (zob. dodatek A), przy tworzeniu języka należało uwzględnić odporność na defekty programu, przy czym wymaganie to nie powinno kolidować z wymaganiem czytelności programu. Z drugiej strony nasze doświadczenie wskazuje, że im program jest napisany bardziej czytelnie, tym ma mniej błędów, i tym łatwiej je znaleźć. Często też jest tak, że reakcja programu na powstanie pewnej grupy błędów jest taka sama – np. wczytanie liczby spoza dopuszczalnego zakresu, błędnie zapisanej itp. – powinno spowodować ponowienie próby czytania tej liczby poprzedzone odpowiednim komunikatem. Dla ilustracji proponujemy Czytelnikowi lub Czytelniczce porównanie dwóch identycznych funkcjonalnie programów — napisanego bez obsługi wyjątków:

```
1 function Pisz return BOOLEAN is
```

```
2  begin
3    if not Drukuj("Wynikiem jest")then
4      return FALSE;
5    end if;
6    if not Drukuj(Wynik) then
7      return FALSE;
8    end if;
9    if not Drukuj(".") then
10     return FALSE;
11   end if;
12   return TRUE;
13 end Pisz;
```

i przy wykorzystaniu mechanizmu obsługi wyjątków:

```
1  function Pisz return BOOLEAN is
2  begin
3    Drukuj("Wynikiem jest ");
4    Drukuj(Wynik);
5    Drukuj(".");
6    return TRUE;
7  exception
8    when BLAD_DRUKOWANIA =>
9      return FALSE;
10 end Pisz;
```

Ponadto, obsługa wyjątku BLAD_DRUKOWANIA może znajdować się nie w procedurze Pisz, ale w innej dynamicznie otaczającej ją procedurze.

Uwaga!

Jeżeli żadna z tych dynamicznie otaczających procedur nie zawiera obsługi tego wyjątku, do jego obsługi przystąpi biblioteka systemowa, która wywołuje procedurę stanowiącą program główny użytkownika – czyli go otacza dynamicznie, w której standardowa obsługa wszystkich wyjątków polega na zakończeniu działania programu głównego lub zadania. O różnicach w obsłudze wyjątku w programie głównym i w zadaniu więcej napisane będzie w rozdziale 10.9. Blok programowy (tj. funkcja (rozdział 6.6), procedura (rozdział 6.8), operator (rozdział 6.7), instrukcja złożona (rozdział 4.3), zadanie (rozdział 14), bariera (rozdział 14.5), instrukcja accept (rozdział 14.6) *dynamicznie otacza inny blok programowy*, jeśli wywołuje kod tego programu. Blok programowy jest *statycznie otoczony przez inny blok programowy*, jeśli instrukcje w nim zawarte nie są widoczne poza zasięgiem widoczności tego bloku. Jeżeli procedura A wywołuje procedurę B, to znaczy, że otacza ją dynamicznie, a jeżeli procedura B jest zadeklarowana wewnątrz procedury A tzn, że procedura A otacza ją statycznie.

Wszystkie takie sytuacje, które nie powinny się zdarzyć w czasie normalnego działania programu nazywane są w Adzie *wyjątkami*. Na ogół sygnalizują one odmienne od „normalnego" zachowanie się programu, i powodują zmianę „normalnej" ścieżki logicznej wykonywania programu. Znane są dwa modele reakcji na „wyjątkowe" działanie programu – *kończenie* stosowane w Adzie (ale i w C++, Javie, Delphi), w którym zakłada się, że błąd ma charakter krytyczny i nie ma możliwości powrotu do miejsca wystąpienia błędu, oraz *wznawianie*, w którym program spodziewa się, że procedura obsługi wyjątku naprawi sytuację. Zapewne sądzisz Cztelniku lub Czytelniczko, że to drugie rozwiązanie jest

lepsze... Ale wyobraź sobie, że operacją, która spowodowała zgłoszenie wyjątku było dzielenie przez zero. Sposobem poprawienia sytuacji byłaby zamiana zera na coś innego. Tylko co innego? A jeżeli tą operacją byłaby próba otwarcia nieistniejącego pliku to jak to naprawić?. Jak widać z tych przykładów – lepiej „dać sobie spokój" i nie naprawiać niczego, zakończyć dany blok programowy (blok declare, podprogram, blok accept) i wtedy wykonać działania zapobiegające wpływowi skutków powstałego błędu na działanie programu.

W Adzie istnieje kilka wyjątków zdefiniowanych pierwotnie (zob. rozdział 10.7), wiele zdefiniowanych jest w pakietach standardowych. Programista może też zdefiniować dowolną liczbę wyjątków odpowiadających innym sytuacjom wyjątkowym. To ostatnie zdanie można zrozumieć dosłownie. Owe sytuacje wyjątkowe mogą odpowiadać błędom programowym, mogą też określać zdarzenia w otoczeniu komputera np. „zawór nie reaguje na sygnał sterujący", ale mogą też odpowiadać „legalnym" sytuacjom, które zdarzają się rzadko i nie warto ze względów efektywnościowych, czy też ze względu na aspekt „czytelności"[1] programu sprawdzać poprawności wykonywania niektórych operacji. Porównaj:

Bez użycia wyjątków

```
...
while not EOF() loop
  Get( Linia_Tekstu );
  Przetwarzaj_tekst();
end loop
...
```

i z ich użyciem

```
...
loop
  Get( Linia_Tekstu );
  Przetwarzaj_tekst();
end loop
...
exception
  when KONIEC_PLIKU =>
  ...
```

10

W pierwszym przypadku, warunek końca pliku będzie sprawdzany bardzo wiele razy, w drugim, gdy system wykryje pojawienie się końca pliku, sterowanie zostanie przekazane od razu do segmentu obsługi, który wcale nie musi znajdować się w tej samej procedurze co pętla loop.

Niektóre wyjątki są zgłaszane przez bibliotekę systemową (np. próba dostępu do nieistniejącego elementu tablicy), ale większość z nich zgłaszana jest w programie całkowicie jawnie po wykryciu odpowiedniego stanu programu.

Z każdym wyjątkiem jest związany co najmniej jeden segment obsługi tego wyjątku. Najbliższy w zagnieżdżonej strukturze dynamicznej programu podejmuje działanie w momencie zgłoszenia wyjątku, przy czym ta część kodu, która powinna być wykonywana po tej instrukcji, która spowodowała zgłoszenie wyjątku jest ignorowana.

[1] To bardzo ważne!

Segment obsługi wyjątku znajduje się zawsze na końcu bloku a jego wykonanie ma taki skutek jak normalne zakończenie tego bloku.

Segmenty obsługi wyjątku są traktowane tak samo jak inne dynamiczne struktury blokowe w programie. W każdym segmencie obsługi mogą być wywoływane inne procedury, które również mogą zgłaszać wyjątki. Problem ten będzie omówiony szczegółowo w rozdziale 10.5.

Mimo rozbudowanych mechanizmów Ady pozwalających pisać programy z mniejszą liczbą błędów, niż takie same programy napisane w innych językach, nie należy się spodziewać, że wszystkie błędy zostaną wykryte przez kompilator lub biblioteki systemowe. Poprawnie napisany program powinien zawierać w wybranych miejscach programu tzw. asercje, czyli warunki określające poprawność logiczną stanu programu w pewnym miejscu. Deklaracja asercji może wyglądać następująco[2]:

```
1 BLAD_ASSERCJI : exception;
2 procedure ASERCJA(WARUNEK : BOOLEAN) is
3 begin
4    if not CONDITION then
5       raise BLAD_ASERCJI;
6    end if;
7 end ASERCJA;
```

a sprawdzenie poprawności logicznej może być następujące:

```
ASERCJA(NAPIECIE >= 1.0 and NAPIECIE <= 10.0);
```

10.1 Deklarowanie wyjątków

Poza kilkoma wyjątkami zdefiniowanymi pierwotnie (zob. rozdział 10.7), każdy wyjątek należy zadeklarować:

```
KONIEC_PLIKU : exception;
BLAD_DRUKOWANIA, SKONCZYL_SIE_PAPIER : exception;
```

10

Mimo, że deklaracja wyjątku do złudzenia przypomina deklarację zmiennych, wyjątek **nie jest zmienną** i nie może być np. parametrem procedury. Wyjątki można tylko deklarować, zgłaszać i obsługiwać. Zasady widoczności, stosowania instrukcji wiążących (renames rozdział 13.2.1) są takie same jak zmiennych i typów.

W przeciwieństwie do na przykład języka Java (Eckel 2001) w Adzie mogą istnieć wyjątki anonimowe. Jeżeli wyjątek zadeklarowany lokalnie w pewnej procedurze zostanie zgłoszony, ale nie obsłużony, to jego obsługi podejmie się, jak to już wiemy, blok dynamicznie otaczający. Niestety nazwa tego wyjątku jest już niedostępna...; Jedynym sposobem obsługi takiego wyjątku jest sekcja others, która wszystkie wyjątki, których programista nie nazwał czy to dlatego, że nie mógł, czy to dlatego, że mu się nie chciało, ładuje do jednego worka, zmuszając do jednakowej metody obsłużenia wszystkich tych – być może skrajnie odmiennych – wyjątków. W języku Java jeżeli procedura zgłasza wyjatek, to informacja

[2] W języku Ada istnieje odpowiednia instrukcja dla kompilatora (pragma) służąca temu celowi. Zainteresowanych Czytelników odsyłamy do (Huzar i inni 1998)

o tym musi się znajdować w jej nagłówku, co powoduje, że jeżeli programista zapomni o obsłużeniu jakiegoś wyjątku, to kompilator przypomni mu o tym fakcie. O tym, że jest to istotne świadczy awaria rakiety Ariane 5 powstała z takiego właśnie powodu (Kopetz 1998).

Trzeba jednak stwierdzić, że w języku Ada, w przeciwieństwie do Javy można wywoływać procedury poprzez wskaźniki do procedur, istnieją też procedury o parametrach klasowych[3] (zob. rozdział 12), których wybór jest dokonywany w czasie działania programu, i które mogą zgłaszać różne wyjątki, niekoniecznie znane w bloku wywołującym tę funkcję.

Dlatego zwracamy się do Czytelników z apelem – słowo komentarza przy nagłówku procedury opisujące co ta procedura robi, jakie jest znaczenie jej parametrów, jakie zgłasza wyjątki i jakie efekty uboczne generuje tak niewiele kosztuje, a tyle czasu oszczędza. Niestety większość programistów (powiedzmy raczej „niedzielnych" programistów), z uporem godnym lepszej sprawy starannie pomija zasady stosowania dobrego stylu programowania. **Czytelniku i Czytelniczko! – bądźcie mądrzejsi!**.

10.2 Zgłaszanie wyjątków

Każdy wyjątek, zarówno zdefiniowany pierwotnie jak i zadeklarowany w programie, może być zgłoszony w zasięgu swojej widoczności instrukcją raise. Na przykład procedura obsługująca drukarkę na podstawie odczytu elektrycznego stanu portu może zgłosić wyjątek poprzez:

```
raise SKONCZYL_SIE_PAPIER;
```

Najczęściej instrukcję tę umieszcza się w warunkowej części programu (po warunku wykrywającym tę sytuację). Dla każdego wyjątku można użyć dowolnie dużo instrukcji raise. Ponadto *wewnątrz segmentu obsługi wyjątków* można użyć specjalnej wersji tej instrukcji:

```
raise
```

oznaczającej powtórne zgłoszenie tego samego wyjątku, który spowodował wejście do tej sekcji obsługi, nawet jeśli go nie można nazwać (np. dlatego, że jest poza zakresem widoczności). W ten sposób można rozłożyć obsługę wyjątku na np. bardziej szczegółową wersję w pewnej procedurze i bardziej ogólną w innej procedurze.

Chociaż wyjątki mogą być jawnie zgłaszane (tj. przez instrukcję raise NAZWA_WYJATKU) tylko w zasięgu widoczności swoich identyfikatorów, może wystąpić niejawne zgłoszenie poza tym zasięgiem wskutek propagacji.

10.3 Obsługa wyjątków

Z chwilą zgłoszenia wyjątku program kończy wykonywanie aktualnego ciągu instrukcji i przystępuje do wykonania odpowiedniego segmentu obsługi. Strefa

[3] Odpowiedniki metod wirtualnych z C++

wyjątków w bloku, składająca się z pewnej (niezerowej) liczby segmentów obsługi wyjątku, umieszczona jest zawsze na końcu bloku i jest poprzedzona słowem kluczowym exception występującym zawsze po ostatniej normalnie wykonywanej instrukcji bloku. Żadna instrukcja segmentów obsługi wyjątku nie jest wykonywana w przypadku, gdy wyjątki nie zostały zgłoszone.

Segmenty obsługi wyjątków wyglądają podobnie jak warianty w instrukcji case (rozdział 4.6, strona 86), z nazwami wyjątków jako selektorami. Nic nie stoi na przeszkodzie, by istniało kilka segmentów obsługi tego samego wyjątku, pod warunkiem, że każdy z nich dotyczy innego bloku (czyli np. innej procedury).

Każdy segment obsługi jest wprowadzony klauzulą when poprzedzającą nazwę wyjątku (lub wyjątków oddzielonych znakiem „|"). Po nazwach następuje znak „=>" i ciąg instrukcji realizujących obsługę tego wyjątku (lub wyjątków) np.:

```
exception
  when KONIEC_PLIKU =>
    null;
  when BLAD_DRUKOWANIA | SKONCZYL_SIE_PAPIER =>
    -- operacje zwiazane z błędem drukarki
  when others =>
    Put_Line("Pojawil_sie_nieznany_wyjatek");
end;
```

Ostatni segment obsługi w bloku można wprowadzić poprzez słowo kluczowe others oznaczające tu „wszystkie inne wyjątki, nawet te, których nie da się nazwać". Ponieważ jednak może się zdarzyć, że co prawda segment obsługi będzie dla „innych" wyjątków taki sam, ale interesująca jest informacja, jaki to wyjątek spowodował wejście do tego segmentu obsługi. Gałąź taka może mieć następującą formę:

```
when Zdarzenie : others =>
  Put_Line( "Pojawil_sie_wyjatek_" & Exception_Name( Zdarzenie ));
```

Przy czym Zdarzenie (typu Exception_Occurence pozwala na identyfikację wyjątku. Jest to jedyna różnica w stosunku do wersji podstawowej segmentu obsługi wyjątków. Dalsze informacje na ten temat znaleźć można w rozdziale 10.6. Oczywiście taka konstrukcja dotyczyć może każdego innego wyjątku, nie tylko wyjątku others.

10

Chcielibyśmy podkreślić, że w języku Ada, w przeciwieństwie do języków Java, C++, Delphi, gdzie tylko część kodu znajdująca się w bloku try jest poddana testom na wystąpienie wyjątków, takim testom poddawana jest *każda* linia kodu.

```
1 try{
2 ...
3 }catch (typ id){
4     ...
5 }catch (typ id){
6     ...
7 }
```

Z tego samego względu w Adzie[4] nie istnieje blok

[4] W zasadzie taki blok nie istnieje także w języku C++, ponieważ „sprzątanie" powinno być zapewnione przez destruktory (zob. rozdział 12), ale w bardzo rozpowszechnionym dialekcie C++ → Visual C++ stanowi on rozszerzenie dostarczone przez Microsoft.

```
try{
...
}finally{
    ...
}
```

którego zadaniem jest „posprzątanie” po działaniu bloku try. Jednakże taki mechanizm wydaje się być nadmiarowy. Ponadto umożliwia on powstanie ryzyka bardzo poważnego błędu polegającego na „zgubieniu” wyjątku (Eckel 2001, p.409).

Zainteresowanych Cytelników zapraszamy do lektury pracy dyplomowej (Buchwald 2001), której treścią jest automatyczny translator z języka Delphi do języka Ada'95 i w której dyskutowane są szeroko odpowiednie konstrukcje języka.

10.4 Przykład użycia wyjątków

```
declare
  Wsp_Delta : FLOAT;
begin
  Wsp_Delta := b**2-4.0*a*c;
  X1 := (-b + sqrt(Wsp_Delta))/(2.0*a);
  X2 := (-b - sqrt(Wsp_Delta))/(2.0*a);
exception
  when ADA.NUMERICS.AUX.ARGUMENT_ERROR:
    if Wsp_Delta >= 0.0 then
      raise; -- wyjatek zgloszony z innego powodu niż ujemy wyroznik
    else
      Put_Line("Rownanie_ma_pierwiastki_zespolone" );
      X1 := 0.0;
      X2 := 0.0;
    end if;
end;
```

W przypadku, gdy wartość zmiennej Wsp_Delta jest ujemna, procedura biblioteczna Sqrt zgłosi błąd ADA.NUMERICS.AUX.ARGUMENT_ERROR. Wartość zmiennych X1 i X2 **nie** zostanie obliczona. Odpowiedni segment obsługi wyjątku po upewnieniu się, że wyjątek ADA.NUMERICS.AUX.ARGUMENT_ERROR został zgłoszony rzeczywiście z tego względu wypisuje odpowiedni komunikat i zeruje zmienne X1 i X2. Jeżeli wyjątek ten został zgłoszony z innego powodu jego obsługa zostanie przeniesiona do procedury otaczającej. W przypadku, gdy współczynnik a jest równy zero zgłoszony jest wyjątek CONSTRAINT_ERROR.

10.5 Propagacja wyjątków

Jak wspomniano wcześniej w momencie zgłoszenia wyjątku, niezależnie od tego czy to zostało zrobione jawnie (instrukcją raise) czy też automatycznie (przez wykrycie błędu wykonania), dalsze wykonywanie bloku, który spowodował zgłoszenie wyjątku zostaje zaniechane i system przechodzi do poszukiwania

odpowiedniego segmentu obsługi. Po dopasowaniu tego segmentu, wykonywany jest ciąg instrukcji związany z tym wyjątkiem.

Początkowo segment obsługi poszukiwany jest w bloku aktualnie wykonywanym. Jeżeli taki segment nie zostanie znaleziony (np. dlatego, że dany blok nie zawiera strefy obsługi wyjątku) lub jeśli segment obsługi sam wykryje wyjątek, to powoduje to zgłoszenie tego wyjątku w bloku dynamicznym otaczającym blok zawierający wyjątek (np. w procedurze otaczającej tę, w której nastąpiło zgłoszenie wyjątku). Takie zachowanie nazywa się propagacją wyjątku. Procedura taka jest wykonywana aż do napotkania głównej procedury programu i jeżeli i w niej nie zostanie znaleziony właściwy segment obsługi wyjątku, to program zostanie zakończony.

Zasadę tę można zawrzeć w następujących punktach:

1. Jeżeli wyjątek jest zgłoszony w ciągu instrukcji stanowiących część podprogramu, to zostanie on przekazany do bloku dynamicznie otaczającego ten ciąg, czyli bloku zawierającego wywołanie tego podprogramu tj. funkcji, procedury lub operatora w wyrażeniu.

2. Jeżeli wyjątek jest zgłaszany w deklaracji (np. przy inicjowaniu zmiennych) to podlega propagacji z bloku zawierającego tę deklarację do instrukcji lub deklaracji, która go wywołała, aż do napotkania instrukcji. Dalsze działanie jest takie jak w punkcie 1.

3. Jeżeli wyjątek zgłaszany jest w obszarze inicjowania pakietu, w którym nie ma lokalnego segmentu obsługi, to jest to równoznaczne z wystąpieniem wyjątku w deklaracji tego pakietu i podlega propagacji do bloku otaczającego.

4. Jeżeli wyjątek zostanie obsłużony, to blok, którego częścią składową jest wykonywana sekcja obsługi wyjątków zostaje uznany za wykonany.

10

10.6 Pakiet Ada.Exceptions

Czasami, choć niezbyt często, zachodzi potrzeba związania z danym wyjątkiem dodatkowych informacji, np. kolekcjonowanie wyjątków, czyli zapisywanie ich w pliku w celu późniejszej analizy funkcjonowania programu. Do tego celu służy stanadrdowy pakiet biblioteczny Ada.Exceptions. Ponadto, z każdym zgłoszeniem wyjątku można związać pewien napis lub inne informacje zależne od implementacji. Przykład:

```
declare
    Szczur: exception;
begin
...
    Raise_Exception (Szczur'Identity, "Niesmaczny_serek");
...
    Raise_Exception (Szczur'Identity, "Trucizna");
...
exception
    when Zdechl : Szczur =>
```

```
        Put ("Powodem zdechnięcia szczura jest =>");
        Put (Exception_Message (Zdechl));
end;
```

Deklaracja omawianego pakietu jest następująca:

```
1  package Ada.Exceptions is
2    type Exception_Id is private;
3    Null_Id : constant Exception_Id;
4    function Exception_Name(Id : Exception_Id) return String;
5    type Exception_Occurrence is limited private;
6    type Exception_Occurrence_Access is access all Exception_Occurrence;
7    Null_Occurrence: constant Exception_Occurrence;
8
9    procedure Raise_Exception(E : in Exception_Id;
10                             Message : in String := "");
11   function Exception_Message(X : Exception_Occurrence) return String;
12   procedure Reraise_Occurrence(X : in Exception_Occurrence);
13
14   function Exception_Identity(X : Exception_Occurrence)
15                                                  return Exception_Id;
16   function Exception_Name(X : Exception_Occurrence) return String;
17   -- Jest to rownowazne Exception_Name(Exception_Identity(X)).
18   function Exception_Information(X : Exception_Occurrence) return String;
19
20   procedure Save_Occurrence(Target : out Exception_Occurrence;
21                             Source : in Exception_Occurrence);
22   function Save_Occurrence(Source : Exception_Occurrence)
23                                        return Exception_Occurrence_Access;
24 private
25   ...
26 end Ada.Exceptions;
```

10.7 Wyjątki zdefiniowane pierwotnie

Każda implementacja Ady definiuje następujące wyjątki:

CONSTRAINT_ERROR Wyjątek ten jest zgłaszany, gdy przekraczane jest ograniczenie takie jak:

- ⇝ Numeryczne przekroczenie zakresu (liczba za duża lub za mała).
- ⇝ Dzielenie przez zero.
- ⇝ Przekroczenie zakresu zmiennej okrojonej (także dostęp do nieistniejącego elementu tablicy).
- ⇝ Dostęp do niewłaściwego wariantu.
- ⇝ Dostęp do zmiennej wskazywanej przez null.

PROGRAM_ERROR Jest zgłaszany przez system w chwili pojawienia się błędnego stanu programu takiego jak np. wywołanie procedury przed inicjacja pakietu czy błędna konkretyzacja ogólnego pakietu programowego (rozdział 13) lub zamknięte wszystkie dozory w instrukcji select (rozdział 14.7).

STORAGE_ERROR Jest zgłaszany, gdy nie powiedzie się wywołanie instrukcji new ze względu na brak pamięci, a także w wyniku przepełnienia stosu (np. w wyniku błędnego napisania procedury rekurencyjnej – rozdział 7).

TASKING_ERROR Jest zgłaszany w przypadku błędu związanego ze spotkaniem.

W niektórych implementacjach mogą być inne pierwotnie zdefiniowane wyjątki.

10.8 Sprawdzanie poprawności w czasie wykonania programu

Ze względu na efektywność kodu konieczne jest czasami wyłączenie sprawdzania poprawności wykonania wstawiane przez kompilator do generowanego kodu. Należy jednak, w miarę możliwości, unikać takich sytuacji[5]. Zrealizowane jest to za pomocą tzw. pragmy, czyli instrukcji dla kompilatora. „Ważność" pragmy kończy się tak samo jak zasięg widoczności zmiennych. Możliwe są dwa typy deklaracji pragmy Suppress sterującej wstawianiem instrukcji testujących przez kompilator.

```
pragma Suppress(Access_Check);
pragma Suppress(Access_Check, On => Wskaznik_Do_Pewnego_Typu);
```

Pierwsze z powyższych użycie pragmy wyłącza sprawdzanie dostępu do zmiennej przez pusty wskaźnik w pewnym ciągu instrukcji, drugie dotyczy tylko zmiennych typu Wskaznik_Do_Pewnego_Typu.

Standardowo wszystkie testy są włączone tak, że większość błędów jest wychwytywana.

Poniżej znajduje się lista parametrów pragmy Suppress:

Access_Check Zgłasza CONSTRAINT_ERROR w przypadku odwołania do zmiennej wskazywanej przez null.

Accessibility_Check Zgłasza PROGRAM_ERROR przy próbie dostępu do niedostępnego obiektu lub podprogramu.

Discriminant_Check Zgłasza CONSTRAINT_ERROR] przy próbie dostępu do niewłaściwego składnika w rekordzie z dyskryminantami.

Division_Check Zgłasza CONSTRAINT_ERROR przy dzieleniu przez zero.

Elaboration_Check Zgłasza PROGRAM_ERROR przy dostępie do opracowanego w niewłaściwej kolejności pakietu lub podprogramu, np. przy próbie wykonania procedury nieopracowanej (np. procedury z pakietu, który nie wykonał części inicjacyjnej).

Index_Check Zgłasza CONSTRAINT_ERROR przy indeksie tablicy spoza zakresu.

[5] Współcześnie procesory są na tyle szybkie, że przyspieszenie działania programu wynikające z zaniechania sprawdzania jego poprawności jest doprawdy trudno dostrzegalne.

Length_Check Zgłasza CONSTRAINT_ERROR przy kopiowaniu tablic o nieodpowiadających sobie wielkościach.

Overflow_Check Zgłasza CONSTRAINT_ERROR przy przekroczeniu zakresu zmiennej numerycznej.

Range_Check Zgłasza CONSTRAINT_ERROR przy zmiennej skalarnej spoza zakresu.

Storage_Check Zgłasza STORAGE_ERROR, jeśli brakuje pamięci przy wywołaniu new.

Tag_Check Zgłasza CONSTRAINT_ERROR, jeżeli obiekt ma niewłaściwą postać.

Accessibility_Check Zgłasza PROGRAM_ERROR, jeżeli obiekt jest niedostępny.

10.9 Obsługa wyjątków w zadaniach

W zasadzie obsługa wyjątków w zadaniach nie różni się niczym od obsługi wyjątków w głównym programie. Jeżeli pewne zadanie, w tym także program główny wykona błędną operację (i nie obsłuży zgłoszonego wyjątku), to zostanie zakończone jego wykonywanie. Jeżeli zadanie to nie ma zadań potomnych (rozdział 14), lub potomne zadania są zakończone, bądź gotowe do zakończenia, to błędne zadanie zostaje zakończone. Jeżeli tym zadaniem jest program główny to zostanie zakończony z odpowiednim komunikatem. Jeżeli jednak pewne zadanie (także program główny) ma zadania potomne, które działają poprawnie, to będą one kontynuowane.

Jeżeli wyjątek zdarzy się w instrukcji accept to segment obsługi wyjątków nie musi znaleźć się w instrukcji złożonej wewnątrz tego bloku, ale może stanowić uzupełnienie instrukcji accept:

```
accept E do
   ...
   ...
exception
   ...
end E;
```

Ponadto, jeżeli wyjątek zostanie zgłoszony wewnątrz instrukcji accept i nie zostanie w niej obsłużony to będzie on propagowany do *obu* zadań biorących udział w spotkaniu (rozdział 14).

10.10 Ćwiczenia

◁ Ćwiczenie 10.1 ▷▷▷ Napisz procedurę czytającą z pliku dane liczbowe, zgłaszającą wyjątek w przypadku wyczerpania pliku i w przypadku przeczytania wadliwej liczby.

◁ Ćwiczenie 10.2 ▷▷▷ Napisz procedurę obsługi dla tych wyjątków.

◁ Ćwiczenie 10.3 ▷▷▷ Zadeklaruj tablicę i spróbuj odwołać się do nieistniejącego jej elementu. Należy zadbać o to, żeby indeks w tablicy był zmienną, ponieważ w innym wypadku kompilator oburzy się na etapie kompilacji. Wykonaj ten program. Obsłuż zgłoszony wyjątek.

◁ Ćwiczenie 10.4 ▷▷▷ Zbuduj podprogram o następującym działaniu: Jeżeli pewien wskaźnik ma wartość null to go zainicjuj, w przeciwnym razie oblicz wartość dowolnego pola wskazywanego przez ten wskaźnik. Podprogram napisz w wersji z obsługą wyjątków, jak i bez niej.

Rozdział 11

Więcej o typach danych

Prócz znanych z innych języków typów danych, język Ada oferuje również bardziej wyszukane i rzadziej stosowane ale tym niemniej bardzo użyteczne typy danych. Umiejętność ich stosowania wykracza naszym zdaniem. poza podstawowy kurs programowania i dlatego omawiane są one w tym miejscu.

11.1 Typy z dyskryminantami

Czasami istnieje potrzeba tworzenia typów nie w pełni określonych. Najprostszym z nich jest typ string zdefiniowany jako:

```
type String is array (Positive range <>) of Character;
```

Zmienne tego typu mogą istnieć tylko jako parametry procedur i funkcji i jako wynik funkcji, jednakże **nie mogą** zaistnieć samodzielnie. Deklaracja:

```
A : String;
```

jest nielegalna. Tworząc zmienną konieczne jest podanie dyskryminatora, czyli parametru określającego ten typ w taki sposób, by kompilator mógł określić wielkość pamięci konieczną do istnienia (przechowania) tej zmiennej np:

```
A : String(1..10);
B : String(A'range);
```

W tym drugim przypadku wielkość ta jest określona dynamicznie na podstawie wymiaru napisu A. Nie należy jednak sądzić, że dyskryminanty ograniczają się do typów tablicowych. W języku Ada są one bardzo szeroko stosowane, a o bardziej zaawansowanych technikach przeczyta Czytelnik lub Czytelniczka w następnych rozdziałach.

Tutaj zostanie opisana tylko deklaracja typów rekordowych z deskryptorami, a w szczególności najczęściej spotykana ich wersja znana z innych języków programowania – w Pascalu jest to *rekord z wariantami*, w C – *unia*.

Jaki jest sens stosowania takich zmiennych? Najczęściej jest to oszczędność pamięci, długości komunikatu itp. Rozważmy następujący przykład:

```
type rodzina is array(Natural range <>) of dziecko;
type pracownik (Mężczyzna : Boolean; Liczba_dzieci : Natural ) is record
  Nazwisko : string( 1..32 );
  case Mezczyzna is
    when True =>
      Byl_w_Wojsku : boolean;
    when False =>
      dzieci : rodzina(1..Liczba_dzieci);
  end case;
end record;
```

Użyjmy teraz następujących deklaracji zmiennych

```
a : pracownik(false,7);
b : pracownik(false,0);
c : pracownik(true,77);
```

gdzie a definiuje kobietę o 7 dzieciach, b kobietę bezdzietną, c mężczyznę. Ponieważ kobieta b ma 0 dzieci, to wektor dzieci jest wektorem pustym i każde odwołanie sie do niego spowoduje wystąpienie wyjątku CONSTRAINT_ERROR. Parametr 77 w definicji pracownika c jest bez znaczenia.

Oczywiście dyskryminanty mogą mieć wartości standardowe i dotyczą ich wszelkie inne cechy charakterystyczne parametrów podprogramów przesyłanych w trybie in.

Ogólnie rzecz biorąc stosowanie rekordów z wariantami i unii jest niebezpieczne ze względu na mozliwość omyłkowego użycia pola nieistniejącego w danym wariancie np. c.dzieci(1). W języku Java z tej właśnie przyczyny usunięto taką konstrukcję z języka, natomiast w języku Ada błąd ten będzie wykryty i zgłoszony zostanie wyjątek CONSTRAINT_ERROR.

Uzupełnienie – Różnice pomiędzy dyskryminantami i parametrami.

Z powyższego tekstu Czytelnik mógłby odnieść – mylne niestety – wrażenie, że dyskryminanty stanowią uogólnienie parametrów. Jednakże, mozliwość stosowania parametrów typów jest okupiona poważnym ograniczeniem: **Dyskryminanty mogą mieć typ albo dyskretny, albo wskaźnikowy**. Ograniczenie to jest bardziej dotkliwe tylko w przypadku typów zadaniowych (rozdział 14.9).

11

11.2 Kształtowanie typów zmiennoprzecinkowych

W trosce o przenośność oprogramowania do języka Ada wprowadzono możliwość definiowania własnych typów zmiennopozycyjnych. W pewnej implementacji typ FLOAT ma precyzję 7 cyfr znaczących, w innej np. 12. Pewien program działający bez zarzutu na danych bardziej dokładnych może wykazywać numeryczną niestabilność na danych mniej dokładnych. W takim przypadku programista musi przenosząc oprogramowanie podjąć decyzję czy w

pewnym miejscu programu stosować typ FLOAT, czy może SHORT_FLOAT, a może LONG_LONG_FLOAT.

Aby pozwolić podjąć tę decyzję kompilatorowi stosuje się nastepującą konstrukcję:

```
type Zmienna_Stanu is digits 15;
type Domain is digits 8 range −1.0 .. +1.0;
type Moc is Zmienna_Stanu range 0.0 .. 1000.00;
```

W przypadku zmiennej typu Zmienna_Stanu kompilator dobierze typ zmiennoprzecinkowy w taki sposób, żeby dokładność obliczeń sięgnęła co najmniej 15 cyfr znaczących, natomiast zakres zmienności pozostaje nieokreślony. W pozostałych przypadkach określona jest zarówno precyzja jak i zakres zmienności. Możliwe jest oczywiście określenie tylko zakresu zmienności zmiennych pewnego typu. Jest to szczególnie użyteczne przy wczesnym wykrywaniu pewnych błędów. Np jeśli pewna zmienna, która jest wartością sinusa kąta przekracza 1.0, to z całą pewnością jest to powód do przeanalizowania kodu.

11.3 Typy stałoprzecinkowe

Co to są liczby stałopozycyjne? Otóż są to takie liczby, które mają ściśle określoną dokładność części całkowitej i ułamkowej. Wszystkie operacje na tych liczbach są bardzo podobne do operacji całkowitoliczbowych – są zatem bardzo szybkie i **dokładne**. W obliczeniach finansowych nie wolno stosować liczb zmiennopozycyjnych ze względu na niedokładność obliczeń. Można zatem stosować obliczenia całkowitoliczbowe w najmniejszych jednostkach (tj. w groszach) lub – wygodniej – stałopozycyjne w złotówkach. Ponadto jest to naturalny typ danych odczytywanych z różnych urządzeń. Np. dane odczytywane z 12-bitowego przetwornika analogowo-cyfrowego o zakresie napięć -5.0V .. +5.0V mogą być danymi typu zdefiniowanego następująco:

```
type Dane_AC is delta 10.0/(2**12) range −5.0 .. +5.0;
```

Nie istnieje żaden pierwotnie zadeklarowany typ stałopozycyjny w rodzaju zmiennoprzecinkowego typu FLOAT albo całkowitoliczbowego typu INTEGER.

11.3.1 Typ dziesiętny

Istnieje też inny sposób deklarowania typów stałopozycyjnych często stosowany m.in. w obliczeniach finansowych.

```
type Zlotowki is delta 0.01 digits 9;
```

definiujący obliczenia na złotówkach w zakresie −9_999_999.99 .. +9_999_999.99.

11.4 Typy zadaniowe i chronione

Typy te są opisane wraz z zadaniami (rozdział 14) i obiektami chronionymi – monitorami (rozdział 14.5). Również rozdział 11.5 należy czytać dopiero po

zaznajomieniu się z rozdziałem związanym z zadaniami ponieważ zastosowanie typów modularnych w czasie pisanie programów nie związanych ze sprzętem wydaje się być ograniczone.

11.5 Typy modularne

Bardzo często w konstrukcji oprogramowania używane są liczniki, tj. zmienne, które w pewnych chwilach są zwiększane (zwykle o 1), ale po osiągnięciu maksymalnej wartości licznik się *przewija* czyli wartość tej zmiennej osiąga wartość zero a *nie jest* zgłaszany wyjątek CONSTRAINT_ERROR tak jak to ma miejsce w każdym innym wypadku. Dotyczy to oczywiście każdej innej operacji na zmiennej tego typu - „normalny" wynik tej operacji jest zamieniany na tę wartość modulo wartość określona w definicji typu. Ponadto na typach modularnych można używać operacji logicznych (or, xor, not i and), co nie jest dozwolone dla innych typów całkowitoliczbowych. Przykładem zastosowania (zob. też rozdział 14.5) może być bufor klawiatury zdefiniowany następująco:

```
1   ...
2   wlk : constant integer := 16;
3   type Idx is mod wlk; -- definicja typu modularnego
4   type zakres is array (idx) of character;
5
6  protected Bufor is
7     entry Get( c : out character );
8     entry Put( c : in character );
9  private
10    Head, Tail : Idx := 0;
11    Ile_zn : integer range 0..wlk-1;
12    Znaki : zakres;
13 end Bufor;
14
15 protected body Bufor is;
16    entry Get( c: out character ) when Ile_zn > 0 is
17    begin
18       c := Znaki(Tail);
19       Tail := Tail+1; -- nigdy nie nastąpi tu przepełnienie
20       Ile_zn := Ile_zn - 1; -- tu kompilator może sprawdzić czy
21                             -- nastąpiło przepełnienie
22    end Get;
23
24    entry Put( c: in character ) when Ile_zn < wlk is
25    begin
26       Znaki(Head) := c;
27       Head := Head+1; -- nigdy nie nastąpi tu przepełnienie
28       Ile_zn := Ile_zn + 1; -- tu kompilator może sprawdzić czy
29                             -- nastąpiło przepełnienie
30    end Put;
31 end Bufor;
```

Deklaracja takiego typu jest zademonstrowana w linii 3 gdzie po identyfikatorze mod następuje wartość, modulo którą traktowany jest wynik. Wartość ta może być całkowicie dowolną dodatnią liczbą całkowitą. W przypadku, gdy liczba ta

jest całkowita potęgą 2, to operacje na takich liczbach wykonywane są znacznie bardziej efektywnie.

Typy modularne odpowiadają typom całkowitoliczbowym w języku C/C++, Java (Stroustrup 1991, Kernighan i Ritchie 1987) Jeżeli Czytelnik lub Czytelniczka zamierza pisać program współpracujący np. z graficzną otoczką (tzw. *GUI*) systemu Microsoft Windows, to stwierdzi, że podstawowe typy danych związane z interfejsem graficznym (DWORD, HANDLE) są typami zdefiniowanym jako

```
type DWORD is mod 2 ** 32;
```

11.6 Zbiory

Niektóre języki programowania (np. Pascal czy Delphi) definiują niezwykle użyteczny typ danych jakim jest typ zbiorowy. W języku C (C++) czy Java (Stroustrup 1991, Kernighan i Ritchie 1987, Eckel 2001) nie można zdefiniować typu zbiorowego, ale zamiast tego stosuje się operacje bitowe na zmiennych całkowitych w taki sam sposób jak w Adzie na liczbach o typie modularnym (zob. rozdział 11.5). Oczywiście nie jest to najszczęśliwsze rozwiązanie, ponieważ wielkość zbioru jest ograniczona do rozmiaru zmiennej typu Integer (ewentualnie Long_Integer) – czyli zwykle są to maksymalnie zbiory 32-elementowe, a poza tym operacje na tych typach są skomplikowane i zaciemniające znaczenie operacji. W tabeli 11.1 znajduje się opis wyrażeń określających podstawowe operacje na zbiorach. Zanim jednak Czytelnik lub Czytelniczka zajrzy do tej tabelki warto przeczytać „jak to się robi".

W Adzie nie ma takiego typu, ale można stworzyć typ zachowujący się identycznie jak typ zbiorowy w oparciu o typ tablicowy. Otóż w Adzie zbiór to tablica, której elementami są wartości logiczne, natomiast typem indeksu jest zbiór elementów typu. Na stronie 41 zdefiniowano typ Dzien_Tygodnia. Odpowiednim zbiorem dni tygodnia będzie:

```
type Dni_Tygodnia is array (Dzien_Tygodnia) of Boolean;
pragma pack(Dni_Tygodnia);
```

Gdyby zabrakło instrukcji dla kompilatora – tzw. pragmy, to kompilator przeznaczyłby dla tablicy taką ilość pamięci jaką uznałby za stosowne – być może różną w różnych implementacjach. Użycie pragmy packed powoduje, że dane zostaną maksymalnie „zagęszczone" czyli dla każdego elementu przeznaczy po jednym bicie.

Dlaczego jednak trzeba włączyć „upychanie" zmiennych? Dlaczego nie jest to standardem? Wynika to z architektury współczesnych procesorów, które znacznie szybciej wykonuja operację jeżeli zmienne na których ta operacja jest dokonywana znajdują sie w miejscu pamięci, którego adres jest podzielny przez 4, a nawet przez 8. Ponadto współcześnie pamięć jest bardzo tania. Dlatego, z powyższych względów opłaca się zostawiać „puste" miejsca w pamięci.

Dla tablic, których elementami są wartości logiczne (i tylko dla takich tablic) zdefiniowane są operatory or, and, xor.

Tabela 11.1: Podstawowe operacje na zbiorach w różnych językach programowania. A i B oznaczają zbiory, E – elementy tych zbiorów.

Operacja	**Zapis matematyczny**	**Pascal**	**Ada**	**Java, C**
Suma zbiorów	$A \cup B$	A+B	A or B	A | B
Różnica zbiorów	$A - B$	A−B	A and not B	A & !B
Iloczyn zbiorów	$A \cap B$	A∗B	A and B	A & B
Dzielenie zbiorów	$(A \cup B) - (A \cap B)$	A/B	A xor B	A ^ B
Negacja zbioru	$\neg A$ lub $\bar{A}$	not A	not A	!A
Dołączenie elementu do zbioru		Incl(A,E)	A(E):= True	A | (1 << E)
Usunięcie elementu ze zbioru		Excl(A,E)	A(E):= False	A & !(1 << E)
Test na przynależność	$E \in A$	E in A	A(E)	A & (1 << E)

11.7 Reprezentacja danych – a jak to wygląda w pamięci

Język Ada zawiera w sobie mechanizmy pozwalające określać szczegóły reprezentacji danych (m.in. wielkość pamięci zajmowaną przez tę daną wyrażoną w **bitach**), szczegóły reprezentacji rekordów, miejsce w pamięci, w którym mają się znaleźć pewne zmienne.

Czytelnik (lub czytelniczka) wiedzą zapewne, że zmienna typu znakowego zajmuje (zwykle) 1 bajt, czyli 8 bitów i można to traktować jako pewnik. No ale ile miejsca w pamięci zajmuje zmienna typu całkowitoliczbowego (Integer)? Tu odpowiedź nie jest już tak prosta, ponieważ w systemach 16-bitowych jest to właśnie 16 bitów, w 32-bitowych $\Rightarrow$ 32. W języku C, czy C++,a nawet w języku Java, jeżeli z pewnych względów zależy nam na tym, żeby dana zmienna była 16 bitowa i przyjmowała wartości z zakresu $-32768.. + 32767$ to nazwiemy ją short int, jeśli przyjmuje wartości nieujemne, to nazwiemy ją unsigned short int itd.

11

A jak to jest w języku Ada? W rozdziale 3 opisany był sposób tworzenia typów okrojonych, tj. z ograniczonym zakresem zmienności, np. definicja type Miesiac is new Integer range 1 .. 12; tworzy typ całkowity (musi to być nowy typ, ponieważ zmiene podtypów mają taki sam rozmiar jak zmienne ich typów bazowych) o bardzo wąskim przedziale zmienności. Łatwo zauważyć, że wszystkie wartości takiego typu można zapisać w 4 bitach a pozostałe (zwykle 28) będą miały wartość równą 0. Toż to czyste marnotrawstwo! Czy jednak dziś, w epoce, w której pamięć jest bardzo tania warto robić takie „groszowe" oszczędności. **Tak!** A jako przykład można podać protokoły transmisyjne, gdzie zmniejszenie ilości niepotrzebnie przesyłanych danych daje znaczące, łatwo zauważalne zyski na efektywności komunikacji.

Wielkość zajętej pamięci dla zmiennej typu Miesiąc można w języku Ada zdefiniować następująco:

```
for Miesiac'Size use 4;
```

Co oznacza, że kompilator zarezerwuje dla zmiennej typu Miesiac tylko 4 **bity**, a wielkość pamięci zajętej przez zmienną w Adzie, inaczej niż w w innych językach programowania określa się w bitach, a nie w bajtach. Jeżeli, zapewne omyłkowo, napisalibyśmy for Miesiac'Size use 3; to oczywiście kompilator zaprotestowałby i uniemożliwił powstanie tego typu błędu.

Jeżeli jednak chcemy „upchnąć" całą datę w jednej 16-bitowej liczbie nie tracąc przy tym nic z zalet Ady (w tym przypadku, ze strukturalności języka), to możemy spróbować napisać następującą definicję:

```
1  procedure td is
2
3      package IIO is new Integer_IO (Integer);
4      use IIO;
5
6      type Miesiac is new Integer range 1 .. 12;
7      type Dzien is new Integer range 1 .. 31;
8      type Rok is new Integer range 1900 .. 2027;
9      for Miesiac'Size use 4;
10     for Dzien'Size use 5;
11     for Rok'Size use 7;
12
13     type Data is record
14         d : Dzien;
15         m : Miesiac;
16         r : Rok;
17     end record;
18
19     d : Data;
20 begin
21     Put (d'Size);
22 end td;
```

Okazuje się niestety, że zmienna d ma wielokość 24 bity! Dlaczego? Przecież $4+5+7=16$ i wszystko powinno się zmieścić w 16 bitach! No tak, ale kompilator ustawia sobie zmienne w pamięci w sposób dowolny i, żeby ułatwić sobie zadanie, każda zaczyna się od granicy całkowitego bajtu (a nawet, w zależności od pragmy Align może się zaczynać od granicy słowa lub nawet podwójnego słowa).

Jak zatem „upchnąć" zmienną w 16 bitach? A może npisać for Data'Size use 16;? Niestety, nie działa. Kompilator upiera się, że minimum to 24. To może pomóc mu i powiedzieć jak ma poukładać sobie pola w tym rekordzie? Napiszmy zatem tak:

```
1      for Data use record
2          d at 0 range 0 .. 4;
3          m at 0 range 5 .. 8;
4          r at 0 range 9 .. 15;
5      end record;
```

Taka deklaracja pozwoliła zapisać rekord na 16 bitach. Na pięciu bitach od 0 do 4 „upchnęliśmy" 5 bitowy numer dnia, na 4 bitach od 5 do 8 $\Rightarrow$ 4 bitowy numer miesiąca itd. Maksymalny numer bitu nie jest ograniczony wielością słowa,

podwójnego słowa, ani w żaden inny sposób (może to być wartość powiedzmy 5791). Ale co oznacza liczba po at? Otóż liczba ta oznacza przesunięcie w bajtach w stosunku do początku rekordu. Możemy zatem nasz rekord zapisać w równoważnej formie:

```
1  for Data use record
2      d at 0 range 0 .. 4;
3      m at 0 range 5 .. 8;
4      r at 1 range 1 .. 7;
5  end record;
```

Co kto lubi, i co uważa za wygodniejsze. Pamiętajmy jednak, że położenie jednobitowego pola (np. typu logicznego, ale nie Boolean, np. type Stan is (On, Off); i dalej for Stan'Size use 1;) również trzeba zadeklarować jako zkres np. s at 0 range 5 .. 5;.

Na koniec przypomnimy, że stałe, określające położenie poszczególnych pól rekordów, ich wielkości itp. mogą być wyrażeniami, ale takimi, żeby kompilator był w stanie obliczyć ich wartość.

Poza określaniem wielkości zmiennej można także wyspecyfikować m.in. bezwzględne położenie zmiennej w pamięci (np. stan pewnej zmiennej systemowej, i inne), ale uważamy, że te cechy języka są zbyt zaawansowane jak na zakres niniejszego skryptu. Zainteresowanych odsyłamy do literatury (Intermetrics Inc. 1995*b*, Intermetrics Inc. 1995*a*, Huzar i inni 1998).

11.8 Ćwiczenia

◁ Ćwiczenie 11.1 ▷▷▷ Zadeklaruj typ rekordowy z dyskryminantami określający obrazek w formacie Windows Bitmap, parametryzowany rozmiarami i liczbą kolorów. Odpowiednie formaty danych znajdziesz w dokumentacji Windows SDK. Zwróć uwagę na wielkość pola informacyjnego. Pamiętaj, że w tym przypadku *nie trzeba* używać wskaźników.

◁ Ćwiczenie 11.2 ▷▷▷ Zadeklaruj typ danych reprezentujących składowe koloru podstawowego i operacje na tym zbiorze danych.

◁ Ćwiczenie 11.3 ▷▷▷ Zadeklaruj odpowiedni typ pozwalający umieścić aktualną godzinę w 16 bitach.

◁ Ćwiczenie 11.4 ▷▷▷ Włóż do nieużywanego portu szeregowego drugą myszkę i czytając z niej znaki spróbuj zgadnąć protokół jakim się ona posługuje. Zadeklaruj odpowiednie typy zmiennych i spróbuj napisać własny sterownik myszy.

Rozdział 12

Programowanie obiektowe

12.1 Co to jest programowanie obiektowe

Wiele lat temu, gdy powstawały pierwsze języki wysokiego poziomu, takie jak Fortran czy Algol, w skład każdego programu wchodziły podprogramy operujące na zmiennych. Zmienne te początkowo miały charakter zmiennych globalnych (czyli dostępnych każdej procedurze — czy ktoś jeszcze pamięta Basic np. na ZX Spectrum?), następnie zmiennych lokalnych i globalnych. Pojawienie się zmiennych lokalnych, czyli takich, które znane są tylko jednej procedurze i znikają po wykonaniu ostatniej instrukcji tej procedury było milowym krokiem w konstrukcji oprogramowania. Podobnym milowym krokiem była możliwość tworzenia zmiennych strukturalnych – głównie rekordów, bo tablice istniały od „zarania dziejów".

Współcześnie pojawiło się bardzo modne, nowe, zainspirowane językiem Simula 67 (Oktaba i Ratajczak 1980), potem językiem Smalltalk podejście do programowania nazywane obiektowym stylem programowania, stanowiące kolejny krok milowy w konstrukcji oprogramowania. Jaka jest podstawowa różnica pomiędzy tradycyjnym, opartym o procedury, stylem programowania a stylem obiektowym? W końcu w dalszym ciągu każdy program składa się ze zmiennych i podprogramów, a i prawdą jest, że przy pomocy tradycyjnego (unikamy tu słowa „starego") stylu programowania można napisac każdy program. Powyższe zdanie można przecież napisać o każdym „milowym kroku" w konstrukcji oprogramowania.

Najogólniej rzecz ujmując różnica polega tylko na stylu programowania i, z pewną przesadą, rzec by można, że tak jak *wygodniej* jest pisać program w języku wysokiego poziomu niż w asemblerze[1], tak styl obiektowy tym góruje nad proceduralnym, że programy wykorzystujące techniki programowania obiektowego łatwiej napisać, zwykle jest w nich mniej błędów, są bardziej czytelne i łatwiejsze w utrzymaniu.

Z czego to wynika?

[1] w języku właściwym dla danego typu procesora

Przed odpowiedzią na to pytanie należy wyjaśnić, czym jest obiekt. *Obiekt* to pewien zbiór danych (tzw. członków klasy) wraz z podprogramami na nim operującymi zwanymi w nomenklaturze obiektowej *metodami*. Bardzo ważne jest by pamiętać, że *klasą* nazywamy definicję obiektu określającą jego strukturę, natomiast sam obiekt jest związany z pewnym obszarem pamięci komputera. Wynika z tego wniosek, że różniących się obiektów danej klasy może być wiele, tak jak wiele egzemplarzy (zmiennych) pewnego typu rekordowego.

Czym jest obiekt? Korzystając z pojęć znanych ze standardowego (tj. proceduralnego) sposobu programowania można powiedzieć, że jest to pewien rekord i zestaw procedur operujących na tym rekordzie. Naturalnie można wyobrazić sobie inaczej zdefiniowany zbiór danych i operacje na nim niekoniecznie będące procedurami (np. API MS Windows) – (Petzold i Yao 1997). Jaka jest zatem różnica pomiędzy obiektem a rekordem z odpowiednim zestawem procedur? Albo pomiędzy obiektem a pakietem również zawierającym zmienne i zestaw operujących na nich procedur? Podstawową różnicą jest to, że obiekty są takimi rekordami, które są rozszerzalne tj. można je uzupełnić o pewne pola (w nomenklaturze obiektowej nazywa się to *dziedziczeniem*). Konsekwencją tego jest konieczność istnienia różnych wersji podprogramów operującym na takim, w różny sposób rozszerzanym, rekordzie. Jeżeli pewna procedura wywołuje inną, która ma wiele wersji w zależności od sposobu rozszerzenia rekordu, to wywołana będzie taka procedura, która dotyczy odpowiedniego rozszerzenia tego rekordu. Takie zachowanie w literaturze obiektowej nazywane jest *polimorfizmem*. Ponadto, jak już wspomnieliśmy, obiekt może istnieć w wielu egzemplarzach, natomiast pakiet nie, choć można go rozszerzać na przykład poprzez pakiety potome (zob. rozdział 9.5).

Powyższe sformułowania wraz z wynikającym z nich konsekwencjami wyczerpują (sic!) opis podstawowych różnic pomiędzy obiektowym i proceduralnym stylem programowania. Ci czytelnicy, którym, znając język Java (Eckel 2001), Delphi (Teixeira i Pacheco 2002), czy C++ (Stroustrup 1991), trudno się zgodzić z powyższym sformułowaniem, niech z krytyką tego poglądu zaczekają do przeczytania tego rozdziału do końca.

Czy do stosowania obiektowego stylu programowania niezbędny jest taki język programowania, który zawiera mechanizmy „obiektowe”? Oczywiście nie, ale bardzo to ułatwia. Kto nie wierzy, niech porówna niewątpliwie obiektowy styl pisania programów w systemie Microsoft Windows w oparciu o API i o biblioteki obiektowe MFC, VCL i inne.

12

Czy, oprócz mody, programowanie obiektowe ma jakieś zalety? Czy takiego samego programu jak napisanego w sposób „obiektowy” nie można napisać w sposób „proceduralny”? Oczywiście, że można. Jednakże podstawową zaletą programowania obiektowego jest łatwość tworzenia takich bibliotek, które można rozszerzać w zależności od potrzeb nie zmieniając ani jednego bajtu w bibliotece bazowej. Ponadto procedury te nie muszą posiadać ogromnej liczby parametrów, co jest często spotykane przy „obiektowym” stylu programowania w językach, które nie pomagają w tworzeniu programów w tym stylu. W ten sposób znacząco zwiększa się niezawodność programów składanych z dobrze przetestowanych „klocków”. Ponadto program aplikacyjny korzystający z takich bibliotek można napisać szybko i jest on dość prosty. Jego tworzenie polega

zwykle na dobraniu odpowiednich „klocków" i takim ich zmodyfikowaniu, żeby odpowiadały aktualnym potrzebom programu.

W Adzie różne aspekty programowania obiektowego zrealizowane są poprzez koncepcje rozszerzalnych typów i poprzez ogólne jednostki programowe opisane w rozdziale 13.

12.2 Rozszerzalność typów

Każdy rekord w języku Ada można zadeklarować jako „rekord możliwy do rozszerzenia" poprzez uzupełnienie jego definicji słowem kluczowym tagged (w wolnym tłumaczeniu – znaczony, lub – ze znacznikiem):

```
type Konto is tagged
   record
      Numer_Konta : Natural := 0;
      Stan : Money := 0.00;
      Oprocentowanie : Stopa_Procentowa := 0.05;
      Odsetki : Money := 0.00;
   end record;
```

Tak zadeklarowany rekord (zob. też rozdział 5.3) może być używany dokładnie tak samo jak rekord bez znacznika. Jednakże, w przeciwieństwie do zwykłego rekordu można go rozszerzyć (uzupełnić) np. w następujący sposób:

```
type Historia_Konta is new Konto with
   record
      Stan_Minimalny : Money := 200.00;
      Stan_Maksymalny : Money := 500.00;
      Liczba_Transakcji : Natural := 0;
   end record;
```

lub

```
type Martwe_Konta is new Konto with
   null record;
```

Oczywiście typ Historia_Konta można również rozszerzyć uzupełniając go o kolejne pola. Typ Martwe_Konto również rozszerza typ Konto o rekord pusty. Mimo, że zbiór danych jest w tym wypadku taki sam, to jednak są to różne typy, w szczególności dlatego, że z każdym z nich mogą być związane inne metody (rozdział 12.3).

Takie rozszerzone typy nazywane będą *typami dziedziczącymi* lub *typami pochodnymi*, natomiast typy, które są rozszerzane nazywane będą *typami bazowymi* lub *typami podstawowymi*.

W przeciwieństwie do rekordów zagnieżdżonych, każde pole rekordu jest widoczne w taki sposób, jak gdyby deklarację pól w rekordzie typu Konto wpisać w deklarację typu Historia_Konta. Dla zmiennej Klient typu Historia_Konta możliwe jest użycie składowych Klient.Stan, Klient.Stan_Minimalny.

Deklaracja zmiennej obu opisanych typów jest identyczna z deklaracją „zwykłych" zmiennych rekordowych.

Jak wspomnieliśmy już w rozdziale opisującym typy zmiennych w Adzie (rozdział 3) Ada jest językiem stosującym niezwykle ścisłe reguły dotyczące operacji na zmiennych różnych typów[2] - zwykle nie pozwalając ich po prostu wykonywać (takie podstawienie jest możliwe jeżeli operacja taka została zdefiniowana). Ponieważ typy Konto i Historia_Konta s ą niewątpliwie różne, wydawać by się mogło, że operacje podstawienia pomiędzy zmiennymi tych typów nie są dozwolone. Ze względu na to, że nawet wielkość tych zmiennych jest różna, nie można też wykonać operacji rzutowania[3].

Jednakże reguła zakazująca operacji podstawienia pomiędzy różnymi typami jest w przypadku typów dziedziczonych nieco rozluźniona. Nie można podstawić do siebie zmiennych typu Historia_Konta i Martwe_Konto (ponieważ typy te są jednak różne). Można natomiast wykonać podstawienia pomiędzy typami bazowymi i dziedziczącymi (rozszerzonymi) w następujący sposób.

```
1 Stare_Konto : Konto;
2 Nowe_Konto : Historia_Konta :=
3            (Stare_Konto with 234.56, 250.00,
4             Liczba_Transakcji => 0);
5 Likwidacja : Martwe_Konto :=
6            (Stare_Konto with null record );
```

Wszystkie pola zmiennej Nowe_Konto, które są dziedziczone z typu Konto przyjmują taką samą wartość jak odpowiednie pola zmiennej Stare_Konto, a pola będące rozszerzeniem przyjmują wartości: Stan_Minimalny $\rightarrow$ 234.56, Stan_Maksymalny $\rightarrow$ 250.00, Liczba_Transakcji $\rightarrow$ 0. W przypadku zmiennej Likwidacja mimo, że pola tego rekordu są identyczne z polami zmiennej Stare_Konto, w podstawieniu należy jednak zaznaczyć, że podstawienie dotyczy typu rozszerzonego.

Podstawienie zmiennej typu Historia_Konta i Martwe_Konto do zmiennej typu Konto można wykonać znacznie prościej np.:

```
Stare_Konto := Konto (Nowe_Konto);
```

Mimo podobieństwa zapisu, **to nie jest rzutowanie**. Pola zmiennej typu Nowe_Konto nie będące składowymi zmiennej Stare_Konto zostaną zignorowane.

12.3 Metody

12

Dla każdego ze zdefiniowanych wcześniej typów można utworzyć procedurę wykonującą na zmiennych tego typu pewne działania np.:

[2] Co bardzo irytuje początkujących programistów, ale jest zbawieniem dla przyzwyczajonych do tej cechy języka

[3] Czyli poinformowania kompilatora, że pewna zmienna ma typ inny niż zadeklarowany. W operacjach, w których typ źródłowy i wynikowy są typami dyskretnymi lub wskaźnikami, lub kompilator zna regułę konwersji, operacja te jest wykonana przez instrukcję Zmienna_Typu_Docelowego := Typ_Docelowy (Zmienna_Typu_Zrodlowego); W pozostałych przypadkach musimy skorzystać z bibliotecznej procedury ogólnej Ada.Unchecked_Conversion. Operacje te opisane są w pełni w (Intermetrics Inc. 1995*b*, Intermetrics Inc. 1995*a*, Huzar i inni 1998).

```
procedure Likwiduj_Konto( Konto_Do_Likwidacji: in out Konto );
procedure Likwiduj_Konto( Konto_Do_Likwidacji: in out Historia_Konta );
procedure Likwiduj_Konto( Konto_Do_Likwidacji: in out Martwe_Konto );

function Szukaj_Wlasciciela( Konto_ID: in out Konto ) return Nazwisko;
function Szukaj_Wlasciciela( Konto_ID: in out Historia_Konta ) return Nazwisko;
function Szukaj_Wlasciciela( Konto_ID: in out Martwe_Konto ) return Nazwisko;
```

Dobór procedury w wywołaniu:

```
Likwiduj_Konto( Opis_Konta )
```

Dokonywany jest przez kompilator na podstawie typu zmiennej Opis_Konta w sposób opisany w rozdziale 6.10. Taki sposób przeciążania identyfikatorów nazywany jest *polimorfizmem statycznym*

O ile istnieje być może sens stosowania trzech różnych procedur do likwidacji różnego rodzaju kont, to z całą pewnością nie ma sensu stosowanie trzech różnych funkcji poszukiwania nazwiska właściciela konta. Możliwe jest takie napisanie programu, żeby jedna procedura była używana dla wszystkich typów pochodnych. Najprostszym rozwiązaniem jest zadeklarowanie funkcji operującej na typie bazowym:

```
function Szukaj_Wlasciciela (Konto_ID: in out Konto) return Nazwisko;
```

wywołanie której może być następujące:

```
Szukaj_Wlasciciela (Konto (Nowe_Konto));
```

Niestety takie rozwiązanie ma wadę polegającą na tym, że ginie bezpowrotnie zawartość pól zmiennej Nowe_Konto, które nie są polami rekordu Konto, oraz ginie informacja o typie zmiennej.

A czy to w czymś przeszkadza? Tak! Na przykład jeżeli w funkcji Szukaj_Wlasciciela należy wywołać odpowiednią dla typu zmiennej Nowe_Konto procedurę Likwiduj_Konto. Skoro utraciliśmy informację o typie – nic nie wiemy o polach rozszerzających.

Takie zachowanie się programu, które pozwala na uzyskanie w trakcie jego wykonywania informacji na temat typu, a przez to pozwalające na wywoływanie odpowiednich podprogramów, i przekazywanie do nich parametrów (także takich, których nasza procedura nie zna ponieważ operuje tylko na typie bazowym), nazywane jest *polimorfizmem dynamicznym*.

Aby skorzystać z opisanego mechanizmu należy funkcję Szukaj_Wlasciciela zdefiniować następująco:

```
function Szukaj_Wlasciciela( Konto_ID: in out Konto'Class )
                             return Nazwisko;
```

Z każdym rekordem, który można rozszerzyć związany jest atrybut Class, który oznacza „wszystkie typy dziedziczące dany typ, łącznie z nim samym”. W tej książce, dla typów z atrybutem Class, używane będzie pojęcie *typ klasowy*. Zatem po wywołaniu funkcji:

```
Szukaj_Wlasciciela (Nowe_Konto);
```

zdefiniowanej następująco:

```
function Szukaj_Wlasciciela( Konto_ID: in out Konto'Class )
                                        return Nazwisko is
...
begin
  ...
  Likwiduj_Konto(Konto_ID);
  ...
end Szukaj_Wlasciciela;
```

w wywołaniu procedury Likwiduj_Konto dobór konkretnej implementacji tej procedury jest dokonany na podstawie typu zmiennej Konto_ID. Oczywiście sama procedura Szukaj_Wlasciciela nie ma dostępu do, *de facto* istniejących, pól rozszerzających typ Konto.

Dla programistów C++, deklaracja taka odpowiada deklaracji z języka:

```
virtual Nazwisko Szukaj_Wlasciciela(void);
```

znajdującej się w deklaracji klasy Konto.

12.4 Własności klas typów

Czasami może okazać się konieczne by podprogram mógł określić, jakiego typu jest zmienna przekazana do tego podprogramu jako typ klasowy. W tym celu stosowana jest następująca konstrukcja języka:

```
...
if Konto_ID in Martwe_Konto then
  ...
end if;
...
```

Warunek w instrukcji if jest prawdziwy wtedy i tylko wtedy gdy Konto_ID jest typu Martwe_Konto. Jest nieprawdziwy, gdy Konto_ID jest typem bazowym, lub pochodnym typu Martwe_Konto.

Jak wspomniano – parametry formalne podprogramów i wyniki funkcji mogą być typu klasowego. Można również zadeklarować wskaźniki wskazujące na zmienne typu klasowego. Deklaracja

```
type Wskaznik_do_konta is access Konto'Class;
```

deklaruje wskaźnik do zmiennej dowolnego typu dziedziczącego typ Konto. Trzeba jednak zdawać sobie sprawę z tego, że rzeczywista zmienna wskazywana przez taki wskaźnik może być różna, w zależności od tego jak rekord Konto został rozszerzony.

Zainteresowanych tą kwestią odsyłamy do pracy (Buchwald 2001), opisującej automatyczny translator z języka Delphi do języka Ada95.

12.5 Typy i podprogramy abstrakcyjne

Jeżeli typ bazowy oznaczony jest wyróżnikiem abstract np.:

```
type Konto is abstract tagged
   record
      Numer_Konta : Natural := 0;
      Stan : Money := 0.00;
      Oprocentowanie : Stopa_Procentowa := 0.05;
      Odsetki : Money := 0.00;
   end record;
```

to dotyczą go wszystkie reguły przedstawione wyżej z tym, że nie mogą istnieć zmienne tego typu, a tylko zmienne typów pochodnych. Oczywiście typy pochodne również mogą być typami abstrakcyjnymi.

Z typami abstrakcyjnymi związane są abstrakcyjne podprogramy, tj. takie podprogramy, które mają deklarację określającą parametry i wynik tych podprogramów, ale nie mają zdefiniowanej treści. Treść ta **musi** zostać zdefiniowana przez implementację nieabstrakcyjnych pochodnych tego typu.

Można definiować nieabstrakcyjne procedury, mające parametry typu abstrakcyjnego, nie można natomiast definiować nieabstrakcyjnych funkcji obliczających wynik typu abstrakcyjnego.

12.6 Obiekty i pakiety

Często uważa się, że programowanie obiektowe zapewnia hermetyzację obiektu i operacji z nim związanych. Polega to na udostępnianiu lub nieudostępnianiu innym jednostkom programowym uprawnień do czytania i nadawania wartości pewnym polom obiektu i do uruchamiania pewnych metod[4].

Tak rzeczywiście jest w C++, czy w Javie, ale w Adzie[5] hermetyzacja obiektów jest zapewniona przez mechanizm pakietów (rozdział 9) i nie ma potrzeby zapewniania dodatkowej hermetyzacji związanej z samymi obiektami.

Fragment deklaracji pewnego pakietu może wyglądać następująco:

```
type File is tagged private;
type Directory is new File with private;
type Ada_File is new File with private;
type Ada_Library is new Directory with limited private;
```

Zgodnie z opisem pakietów (zobacz rozdział 9) pola zmiennych typu File, Directory, Ada_File, Ada_Library są niedostępne na zewnątrz pakietu, a zmienne tych typów mogą być tylko podstawiane do siebie i porównywane. Wyjątek stanowią zmienne typu Ada_Library, na których nie można wykonać nawet tych operacji, ponieważ typ ten jest typem ograniczonym.

[4] Rozwiązanie hermetyzacji w Adzie jest bezpieczniejsze niż w języku C++, ponieważ w języku C++ nic nie zmusza programisty do napisania wszystkich zadeklarowanych metod. Jeżeli programista zapomni o napisaniu pewnej metody i okaże się, że w danym programie metoda ta nie była potrzebna to błąd taki zostanie niezauważony. W Adzie jest to niemożliwe.

[5] w języku Delphi także

12.7 Nadzorowanie obiektów

Programiści znający język C++ na pewno zaniepokoili się brakiem w Adzie konstruktorów i destruktorów, czyli metod wywoływanych automatycznie w momencie odpowiednio, tworzenia i usuwania zmiennej tego typu. Istnieje jednak w języku Ada i taka konstrukcja.

W Adzie zdefiniowany jest pakiet Ada.Finalization o następującej specyfikacji:

```
1  package Ada.Finalization is
2    pragma Preelaborate(Finalization);
3    type Controlled is abstract tagged private;
4    procedure Initialize(Object : in out Controlled);
5    procedure Adjust (Object : in out Controlled);
6    procedure Finalize (Object : in out Controlled);
7    type Limited_Controlled is abstract tagged limited private;
8    procedure Initialize(Object : in out Limited_Controlled);
9    procedure Finalize (Object : in out Limited_Controlled);
10 private
11   ... -- nie określone przez język
12 end Ada.Finalization;
```

Jeżeli zdefiniowany przez użytkownika typ jest pochodnym typu bazowego Ada.Finalization.Controlled lub Ada.Finalization.Limited_Controlled to w momencie powołania zmiennej tego typu automatycznie wywoływana jest procedura Initialize pełniąca rolę bezparametrowego konstruktora, której zadaniem jest inicjacja wszystkich pól tej zmiennej. W momencie usuwania tej zmiennej wywoływana jest procedura Finalize pełniąca rolę destruktora, a w przypadku podstawienia po skopiowaniu wszystkich pól wywoływana jest procedura Adjust.

W przypadku kodu:

```
1  Declare
2    Type Element is new Ada.Finalization.Controlled with
3      Record
4          ...
5    end Element;
6    a, b : Element; -- wywoływane jest Initialize(a), potem Initialize(b)
7  Begin
8    ...
9    a := b; -- Finalize(a), potem kopiowanie zawartości, potem Adjust(b)
10   ...
11 end; -- Finalize(b), potem Finalize(a)
```


Standardowa definicja tych procedur Initialize i Finalize sprowadzona jest do operacji „nie rób nic”, zaś procedury Adjust do kopiowania zawartości wszystkich pól odpowiednich zmiennych.

Jednakże trzeba tu wyraźnie powiedzieć, że w przypadku, w którym mamy do czynienia z typem, którego elementem składowym są wskaźniki do typu dziedziczącego po typie Controlled, to odpowiednia procedura Finalize **nie jest** wywoływana w przypadku, w którym zaniknie dostęp do tej struktury, a dopiero po zakończeniu programu. W najprostszym przypadku jeżeli zdefiniujemy następujące pakiety:

```
with Ada.Finalization;

package Test_AC_AC is

   type ACT is new Ada.Finalization.Controlled with
      record
         s : String (1 .. 2);
      end record;


   procedure Finalize (f : in out ACT);

end Test_AC_AC;
```

```
with Text_IO; use Text_IO;

package body Test_AC_AC is
   procedure Finalize (f : in out ACT) is
   begin
      Put_Line (f.s);
   end Finalize;
end Test_AC_AC;
```

i w programie głównym zdefiniujemy najprostszy możliwy typ zawierający typy dziedziczące po typie Controlled, tj. tablicę:

```
with Test_AC_AC; use Test_AC_AC;
with Text_IO; use Text_IO;
with Ada.Unchecked_Deallocation;

procedure Test_AC is
   package IIO is new Integer_IO (Integer);
   use IIO;

   type Tablica_O is array (Integer range <>) of ACT;
   type Tablica_P is access Tablica_O;
   t : Tablica_P;

   procedure Delete is new Ada.Unchecked_Deallocation (Tablica_O, Tablica_P);


begin
   t := new Tablica_O (1 .. 5);
   for idx in t'Range loop
      Put (t (idx).s, idx);
   end loop;
   Delete (t);
   Put_Line ("Koniec");
end Test_AC;
```

to w przypadku, gdyby zamiast linii Delete (t) była linia t = null, to odpowiednia procedura Finalize zostałaby wywołana dopiero po zakończeniu programu. W przypadku przedstawionym w powyższym programie ciąg wywołań procedury Finalize zostanie wywołany jako uboczny skutek procedury Delete.

Konstrukcja wykorzystywana w powyższym programie odpowiada konstruktorowi bezparametrowemu (czasami zwanego też domyślnym). Ale czasami niezbędne jest zainicjowanie obiektu w pewien szczególny sposób, np. w przypadku, w którym z danym obiektem skojarzony jest plik, to do jego zainicjowania niezbędne jest podanie jego nazwy. Zresztą przykłady można mnożyć.

Ponadto możliwe są różne warianty stworzenia obiektu np. w przypadku obiektu będącego opisem pewnego okienka na ekranie, konstruktor bezparametrowy tworzy okno o domyślnych rozmiarach w środku ekranu, ale programista życzy sobie mieć konstruktory tworzące okno w zdefiniowanym położeniu, albo tylko o określonych wymiarach.

W języku Java, czy C++, tworzy się dowolną liczbę procedur o nazwie tożsamej z nazwą klasy (czyli w nomenklaturze Ady – rekordu znakowanego). A w Adzie można napisać funkcję, która oblicza taki, odpowiednio zainicjowany rekord.

Jeżeli zdefiniujemy chociaż jedną taką funkcję, to kompilator zadba już o to, żeby taką samą funkcję zdefiniować dla każdego typu dziedziczącego po tej klasie (No chyba, że klasa zostanie zdefiniowana jako abstrakcyjna, to wtedy ewentualny protest zostanie „odłożony" na później – do czasu zdefiniowania klasy nieabstrakcyjnej).

```
1  package Klasy is
2
3      type Klasa is tagged record
4          a : Integer := 1;
5      end record;
6
7      function Create (a : Integer) return Klasa;
8
9      type Klasa_P is new Klasa with
10         record
11             c : Character;
12         end record;
13     -- Kompilator zaprotestuje, ponieważ nie zdefiniowano
14     -- funkcji Create
15 end Klasy;
```

Użycie tej klasy

```
1  with Klasy; use Klasy;
2
3  procedure KlasyU is
4      k1 : Klasa; -- k1.a = 1
5      k2 : Klasa := Create(100); -- k2.a = 100
6  begin
7      ...;
8  end KlasyU;
```

Niestety kompilatory nie są na tyle sprytne, żeby zapobiec użyciu całkiem niezainicjowanego obiektu, którego pola nie są inicjowane automatycznie i nie istnieje dla niego konstruktor bezparametrowy pochodzący z pakietu Ada.Finalization.

12.8 Wielokrotne dziedziczenie

Pewne języki programowania (np. C++) pozwalają na taką deklarację typu, który ma więcej niż jeden typ bazowy, tj. rozszerza więcej niż jeden typ. Taką możliwość nazywa się *wielokrotnym dziedziczeniem*.

Jest to mechanizm koncepcyjnie znacznie bardziej złożony niż zwykłe dziedziczenie (Stroustrup 1991). Ponadto występują pewne trudności z efektywną implementacją takiego mechanizmu.

Z tych powodów w Adzie zrezygnowano z konstrukcji językowych pozwalających na wielokrotne dziedziczenie. Zamiast tego można stosować kompozycje ogólnych jednostek programowych i rozszerzalnych typów.

12.9 Składanie implementacji i abstrakcji

Powodem, dla którego stosuje się wielokrotne dziedziczenie jest możliwość złożenia pewnego obiektu z obiektów bazowych, które mają zupełnie różne logicznie zastosowania w pewien obiekt. Przykładem może tu być protokół komunikacji i struktury danych. Wprowadzona w rozdziale 12.10 lista dwukierunkowa opisuje pewną strukturę danych nie mającą nic wspólnego z protokołem komunikacyjnym. Jeśli jednak listę tę trzeba przesłać, to należy w tym celu użyć pewnego protokołu opisanego inną strukturą danych (obiektem).

W C++, gdzie klasy są jedyną formą modularyzacji oprogramowania, dziedziczenie jest jedynym sposobem składania abstrakcji. W Adzie istnieje więcej takich sposobów. Jeżeli zadeklarowana zostanie klasa (rekord rozszerzalny):

```
type Stos is abstract tagged null record;
```

i zdefiniowano abstrakcyjne operacje na stosie (Push i Pop) to w przypadku pewnego stosu należy ją rozszerzyć, np. poprzez deklarację, w której następuje konkretyzacja realizacji stosu i nadawana jest wartość zakresowi tablicy decydującej o głębokości stosu:

```
type Pewien_Stos is new Stos with
  record
    Tablica : array (0..10) of T;
  end Pewien_Stos;
```

Jest to oczywiście rozwiązanie poprawne, ale stanowczo zbyt skomplikowane. Znacznie prościej zdefiniować typ:

```
type Stos_Array is array(Integer range <>) of T;
```

i operacje na nim zdefiniowane. Stosowanie typów nie w pełni zdefiniowanych (rozdział 11.1) jest często znacznie prostszą i znacznie bardziej efektywną metodą niż stosowanie obiektów. Szczególnie użycie ogólnych jednostek programowych (rozdział 13) jest niezwykle potężnym narzędziem, którego użycie w dużej mierze może uprościć, a nawet zastąpić użycie mechanizmów programowania obiektowego.

12.9.1 Dziedziczenie mieszane

Konstrukcją, która w niektórych przypadkach z powodzeniem zastępuje wielokrotne dziedziczenie jest jednoczesne użycie obiektów i ogólnych jednostek programowych (rozdział 13) będących rodzajem wzorców (ang. *templates*). Rozważmy następujący przykład:

```
1  generic
2     type S is abstract tagged private;
3  package P is
4     type T is abstract new S with private;
5     -- operacje na typie T
6  private
7     type T is abstract new S with
8        record
9           -- dodatkowe skladniki
10       end record;
11 end P;
```

Naturalnie treść ogólnej jednostki programowej[6] definiuje operacje i specyfikację rozszerzonego typu i funkcji na nim operujących.

Przy takiej deklaracji można utworzyć obiekt R w celu rozszerzenia typu S i taki rozszerzony typ R w dalszym ciągu może służyć jako parametr opisanej jednostki ogólnej, w której jest znów rozszerzany. W zaprezentowanym przykładzie zarówno typ formalny S jak i eksportowany T, zdefiniowany w pakiecie P, są zadeklarowane jako typy abstrakcyjne (rozdział 12.5), z czego wynika wniosek, że nie mogą istnieć zmienne tego typu. W podobny sposób można deklarować kaskadę typów, z których ostatni nie może być już typem abstrakcyjnym.

W opisany sposób jednostka ogólna uzupełnia własności różnych wersji pewnego typu obiektowego, czyli zapewnia pewien sposób wielokrotnego dziedziczenia. Poniżej przedstawiono konkretny przykład zastosowania takiego mechanizmu:

```
1  with OM; -- Object Manager zapewnia unikalny identyfikator obiektu
2  with VM; -- Version Manager zapewnia obsługę wersji
3  generic
4     type Rodzic is abstract tagged private;
5  package Dostarczanie_Wersji is
6     -- Obiekt posiadający jednoznaczny identyfikator, a jednocześnie
7     -- posiadający obsługę wersji tego obiektu.
8     -- Wersja wraz z identyfikatorem jednoznacznie określa obiekt
9     type Obiekt_Z_Wersja is abstract new Rodzic with private;
10    -- na podstawie obiektu oblicza jego nową wersję
11    procedure Tworz_Nowa_Wersje(O : in Obiekt_Z_Wersja
12                                Nowy_O: out Obiekt_Z_Wersja);
13    -- na podstawie obiektu oblicza jego wersję
14    function Numer_Wersji(O: Obiekt_Z_Wersja)
15                                      return VM.Version_Number;
16    -- na podstawie obiektu i numeru wersji oblicza obiekt
17    procedure Znajdz_Obiekt(
18      ID_From: in Obiekt_Z_Wersja;
```

[6] Czasami stosuje się też, naszym zdaniem nieprawidłowy, termin „jednostka rodzajowa”

```
19          Version: in VM.Version_Number;
20          Object : out Obiekt_Z_Wersja);
21    private
22       type Obiekt_Z_Wersja is abstract new Parent with
23          record
24             ID : OM.Obiekt_ID := OM.Unikalny_ID;
25             Version: VM.Numer_Wersji := VM.Wersja_Poczatkowa;
26          end record;
27    end Dostarczanie_Wersji;
```

Zaletą takiej konstrukcji jest możliwość rozszerzenia obiektu „bez wiedzy" jednostki ogólnej. Polega to na tym, że pełen typ (tj. typ z rozszerzeniem prywatnym) może nie być bezpośrednim potomkiem danego przodka. Dlatego pełen typ odpowiadający:

```
type Obiekt_Specjalny is new Przodek with private;
```

może nie rozszerzać bezpośrednio typu Przodek, a może rozszerzać typ pochodny typu Przodek. Dlatego konkretyzując ogólną jednostkę programową można napisać:

```
private
   package Q is new P(Przodek);
   type Obiekt_Specjalny is new Q.T with null record;
```

W takim przypadku typ Obiekt_Specjalny będzie zawierał także wszystkie pola i metody typu T pochodzącego z pakietu ogólnego P. Jest oczywiste, że pola takie nie będą widoczne dla klientów, ale podprogramy zadeklarowane w widocznej części pakietu, w którym zadeklarowany jest Obiekt_Specjalny mogą być używane w zwykły sposób. Należy zauważyć, że typ Obiekt_Specjalny nie jest abstrakcyjny, nawet gdy typ Q.T jest typem abstrakcyjnym.

Innym przykładem mieszanego dziedziczenia jest pakiet zarządzający kolejkami jednokierunkowymi. Kolejką nazywa się taki zbiór danych, który może zawierać dowolną – w tym zerową – liczbę elementów, przy czym jeżeli każdy z nich „zna" tylko następny element tego zbioru (tej listy) to lista taka jest *listą jednokierunkową* (strona 168), a jeżeli zna i element następny i poprzedni to jest to *lista dwukierunkową*. Każdy inny element na liście dostępny jest tylko poprzez przeglądanie kolejnych elementów listy. Taka operacja nazywana jest *iteracją*. Zwykle lista składa się z elementów tego samego typu – jest to tzw. *lista homogeniczna* zwana po prostu *listą*. Jeżeli lista składa się z elementów różnego typu to jest to *lista heterogeniczna* zwana czasami *kolekcją*.

Poniższy przykład pokazuje jak zdefiniować kolekcję, przy wykorzystaniu „zwykłych" tj. nieobiektowych struktur danych. Za pomocą samej ogólnej jednostki programowej uzyskalibyśmy listę homogeniczną.

```
1  generic
2    type Dane(<>) is abstract tagged private;
3  package Kolejki is
4    type Kolejka is limited private;
5    type Element_Kolejki is abstract new Dane with private;
6    type Wskaznik_Do_Elementu is access all Element_Kolejki'Class;
7    function Jest_Pusta(Q: Kolejka) return Boolean;
8    procedure Dodaj_Do_Kolejki(Q: access Kolejka;
```

```
9                                   E: in Element_Kolejki);
10    function Usun_Z_Kolejki(Q: access Kolejka) return Element_Kolejki;
11    Blad_W_Obsludze_Kolejki: exception;
12  private
13    ...
14  end Kolejki;
```

a konkretyzacja tej jednostki przez program korzystający z pakietu Kolejki mogłaby wyglądać następująco:

```
1  with Kolejki;
2  package System_Alarmow is
3    type Alarm is abstract tagged null record;
4    package Kolejki_Alarmow is new Kolejki(Root_Alert);
5    subtype Kolejka_Alarmow is Kolejki_Alarmow.Kolejka;
6    type Pewien_Rodzaj_Alarmu is abstract
7      new Kolejki_Alarmow.Element_Kolejki with null record;
8    procedure Obsluz(A in out Pewien_Rodzaj_Alarmu) is abstract;
9  end System_Alarmow;
```

Z kolei użycie pakietu System_Alarmow mogłoby wyglądać następująco:

```
type Wskaznik_Do_Kolejki_Alarmow is access all Kolejka_Alarmow;
Pewna_Kolejka : Wskaznik_Do_Kolejki_Alarmow := new Kolejka_Alarmow;
...
```

W wersji pakietu obsługującego kolejki, w której typ elementów jest nieokreślony, utworzone kolejki są ogólne[7]. Żaden mechanizm formalny nie stoi zatem na przeszkodzie, by w kolejce alarmów umieścić omyłkowo obiekt jakiegoś innego typu np. opis paczki pobieranej z magazynu. W obiektowej wersji, choć kolejka jest kolejką heterogeniczną, to jednak jest ona ograniczona do obiektów dziedziczących pewną klasę (tj. rozszerzających pewien rekord).

Ostatni przykład pokazuje jak może być skonstruowany system okien i ilustruje kaskadę dziedziczenia mieszanego. Podstawowa definicja okna i operacji na nim zdefiniowanych może być następująca:

```
type Podstawowe_Okno is tagged limited private;
procedure Pokaz(W: in Podstawowe_Okno);
procedure Klikniecie_Myszka(W: in out Podstawowe_Okno;
                            Gdzie: in Wspolrzedne_Polozenia_Myszy);
...
```

Następnie można zdefiniować bazujący na tym typie ogólny pakiet:

```
1    generic
2      type Pewne_Okno is abstract new Podstawowe_Okno with private;
3    package Opisane_Okno is
4      type Okno_Z_Naglowkiem is abstract new Pewne_Okno with private;
5      -- wymiana pewnych operacji zdefiniowanych dla typu
6      -- "Podstawowe_Okno"
7      procedure Pokaz(W: in Okno_Z_Naglowkiem);
8      -- dodanie innych operacji
9      ...
```

[7] Sposób deklaracji m.in. takiej kolejki jest opisany w rozdziale 10.

```
10      procedure Ustaw_Naglowek( W: in out Okno_Z_Naglowkiem;
11                                S: in String);
12      function Naglowek(W: Okno_Z_Naglowkiem) return String;
13  private
14      type Okno_Z_Naglowkiem is abstract new Pewne_Okno with
15          record
16              Naglowek: Odpowiedni_Napis := Pusty_Napis;
17              -- Unikalny nagłówek okna
18          end record;
19  end Opisane_Okno;
```

Procedura Pokaz może być tu zdefiniowana następująco:

```
1   procedure Pokaz(W: Okno_Z_Naglokiem) is
2   begin
3       Pokaz (Pewne_Okno (W));
4       -- Powyzsza instrukcja pokazuje okno w zwykly sposób
5       -- odziedziczony po przodku
6       if W.Naglowek /= Pusty_Napis then
7           -- Jezeli etykieta jest odpowiednia => rysuj okno
8           Pokaz_Na_Ekranie( Wspolrzedna_X(W),
9                             Wspolrzedna_Y(W)-5,
10                            W.Naglowek);
11      end if;
12  end Pokaz;
```

przy czym funkcje Wspolrzedna_X i Wspolrzedna_Y są dziedziczone z typu Podstawowe_Okno i określają współrzędne tego okna.

Nic nie stoi na przeszkodzie by zadeklarować ciąg takich pakietów:

```
1   package Ramka is
2       type Moje_Okno is new Podstawowe_Okno with private;
3       ... -- funkcje eksportowane (widoczne)
4   private
5       package Dodany_Naglowek is new Opisane_Okno(Podstawowe_Okno);
6       package Dodana_Ramka is
7                   new Opisane_Okno(Dodany_Naglowek. Okno_Z_Naglowkiem);
8       package Dodane_Menu is
9                   new Okno_Z_Menu(Dodana_Ramka.Okno_Z_Ramka);
10
11      type Moje_Okno is
12                  new Dodane_Menu.Okno_Z_Menu with null record;
13  end Ramka;
```


12.9.2 Polimorfizm struktur

Zwykle pod pojęciem polimorfizmu rozumie się kolekcję procedur, z parametrem typu T'class, których wybór jest automatycznie dokonywany na podstawie typu zmiennej (zob. rozdział 12.3). *Polimorfizmem struktur* nazywa się taką deklarację struktur, która nie w pełni określa ich postać.

Wykorzystanie tego mechanizmu jest jeszcze jedną prostą metodą symulacji wielokrotnego dziedziczenia korzystającą z mechanizmu dyskryminantów wskaźni-

kowych (rozdział 13.3.1) pozwalającą parametryzować pewien rekord za pomocą innego rekordu.

Dyskryminant wskaźnikowy może być użyty w celu udostępnienia pola w rekordzie w celu uzyskania rekordu, w którym jest zagnieżdżony (zob. rozdział 13.3.1). Pozwala to na tworzenie mechanizmu polimorfizmu struktur.

W następującym przykładzie:

```
type Zewnetrzny is limited private;
private
   type Wewnetrzny(Wskaznik: access Zewnetrzny) is limited ...
   type Zewnetrzny is limited
      record
         ...
         Skladnik : Wewnetrzny (Zewnetrzny'Access);
         ...
      end record;
```

Pole Skladnik typu Wewnetrzny jest parametryzowane dyskryminantem ze wskaźnikiem Wskaznik, który odnosi się do typu dziedziczącego po rekordzie Zewnetrzny, ponieważ atrybut Access nazwy typu rekordowego wewnątrz rekordu odnosi się do aktualnej konkretyzacji typu. Jest to sytuacja podobna do tej, w której nazwa zadania odnosi się do danego zadania (jego treści) zamiast do nazwy typu.

Jeżeli obiekt Zewnetrzny zadeklarowany będzie następująco:

```
Obj: Zewnetrzny;
```

To, całkowicie automatycznie, zostanie utworzona struktura wskazująca na siebie (rys. 12.1)

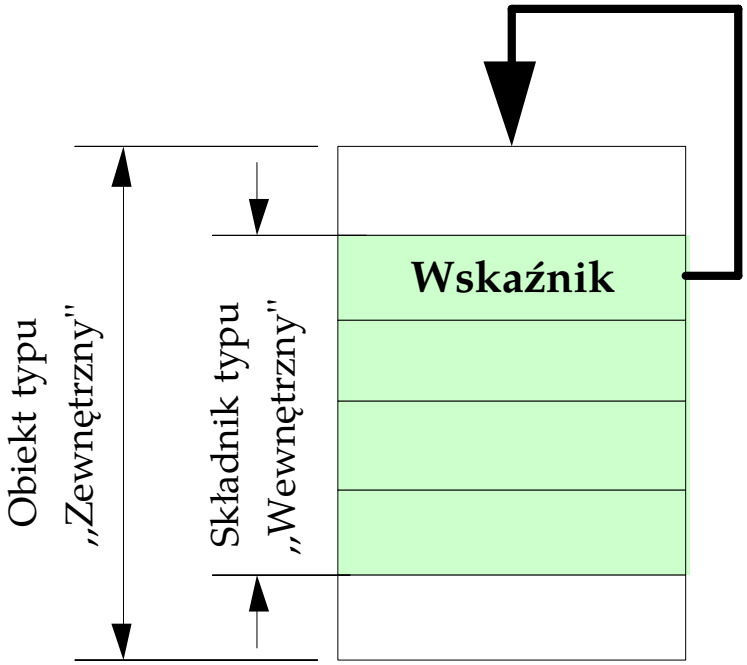
Konieczne jest by zauważyć, że efekt ten nie jest równoważny temu, który można uzyskać poprzez rekord z polem wskazującym na typ tego rekordu (rozdział 8) ponieważ wszystkie konkretyzacje (zmienne) typu Zewnetrzny będą pokazywały na siebie. Pole Wskaznik nie może zostać zmienione, ponieważ dyskryminanty są stałymi.

Oczywiście przedstawiony wyżej prosty przykład jest niezbyt interesujący. Warto tu jednak dostrzec, że typy Wewntrzny i Zewnetrzny mogą być rozszerzeniami innych typów, które mogą być dowolnie złożonymi strukturami. Np. typ Wewnetrzny może być rozszerzeniem typu Wezel zawierającego pola wskazujące na inne obiekty tego typu tworzące np. drzewa. W szczególności typ Wewnetrzny może być także zdefiniowany następująco:


```
type Wewnetrzny(Wskaznik: access Zewnetrzny'Class) is
   new Wezel with ...
```

Wynikiem takiej deklaracji jest utworzenie heterogenicznych związków (oczywiście typ Zewnetrzny musi być w tym wypadku typem obiektowym). W takiej strukturze można przechodzić z jednego węzła (typu Wewnetrzny) do drugiego, a w każdej chwili pole Wskaznik pokazuje na otaczający obiekt typu Zewnetrzny.

Warto zauważyć, że dyskryminanty wskaźnikowe są dozwolone tylko dla typów ograniczonych ze względu na konieczność unikania problemów związanych z



Rysunek 12.1: Struktura odwołująca się sama do siebie

kopiowaniem takich struktur i z wskaźnikami nie odnoszącymi się do żadnej legalnej zmiennej.

Powstaje jednak pytanie, jaki związek mają takie wyrafinowane struktury z wielokrotnym dziedziczeniem?

Przypuśćmy, że należy utworzyć strukturę łączącą dwie hierarchie – obiektów graficznych (Obiekt_Graficzny) i operacji, które taki obiekt pozwalają utworzyć (Operacja_Graficzna). Dla przykładu obiektem graficznym może być czworościan, a operacją graficzną może być narysowanie każdej z jego ścian.

Typ Operacja_Graficzna jest zaprojektowany tak, by odpowiadać na zmiany widoku (np. związane ze zmianą jego położenia) pewnego obiektu graficznego. Jeżeli pewna operacja życzy sobie być zastosowaną (tu: wyrysowaną) na ekranie, to wywoła procedurę Aktualizuj. Oczywiście różnym operacjom odpowiadają różne rozszerzenia typu Operacja_Graficzna. Stąd następujące deklaracje:

```
1  type Operacja_Graficzna;
2  type Wskaznik_Do_Operacji_Graficznej is
3                      access all Operacja_Graficzna'Class;
4
5  type Obiekt_Graficzny is abstract tagged limited
6    record
7      Pierwszy_Element: Wskaznik_Do_Operacji_Graficznej; -- lista obiektów
8      -- pozostałe składniki mogą być dodane przez rozszerzenie typu
9      -- zależnie od konkretnej aplikacji
10   end record;
11
12 type Wskaznik_Do_Obiektu_Graficznego is
```

```
13                              access all Obiekt_Graficzny'Class;
14
15 type Operacja_Graficzna is abstract tagged limited
16   record
17     Nastepny : Wskaznik_Do_Operacji_Graficznej;
18     Obiekt : Wskaznik_Do_Obiektu_Graficznego;
19     -- pozostałe składniki mogą być dodane przez rozszerzenie typu
20     -- zależnie od konkretnej aplikacji
21   end record;
22
23 procedure Aktualizuj(M: in out Operacja_Graficzna) is abstract;
24 ...
25 procedure Zaznacz_Zmiany(GO: Obiekt_Graficzny'Class) is
26   t : Wskaźnik_Do_Operacji_Graficznej:= GO.Pierwszy_Element;
27 begin
28   while t /= null loop -- dopóki nie wykonano operacji dla wszystkich
29     Aktualizuj(t.all); -- dla każdego obiektu Obiekt_Graficzny
30     t := t.Nastepny;
31   end loop;
32 end Zaznacz_Zmiany;
```

gdzie Zaznacz_Zmiany jest operacją na typie klasowym Obiekt_Graficzny (czyli w rzeczywistości na typie dziedziczącym po Obiekt_Graficzny) wywołującą operację Aktualizuj na wszystkich obiektach z listy dziedziczących własności typu Operacja_Graficzna. Jeżeli obiekt opisany typem Obiekt_Graficzny ma reprezentować wielościan, jego deklaracja mogłaby być następująca:

```
1 type Wieloscian is new Obiekt_Graficzny with
2   record
3     M: Parametry_Wieloscianu;
4   end record;
5 ...
6 figura: Wieloscian;
```

Po obliczeniu np. nowego położenia wielościanu, jego obraz na ekranie może być zaktualizowany przez wywołanie:

```
Notify(figura);
```

Powstała struktura może być zobrazowana w sposób pokazany na rysunku 12.2.

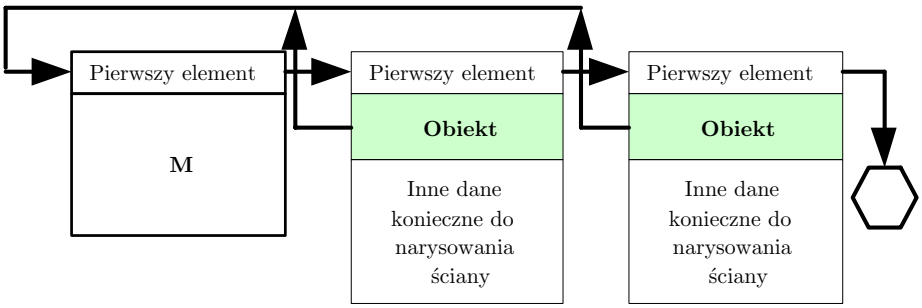
Przypuśćmy, że należy skojarzyć odświeżanie obrazu obiektu graficznego z częścią okna, którego to odświeżanie ma dotyczyć. Wyrażając się ściślej należy utworzyć taki prostokąt, który ma zarówno cechy okna jak i cechy obiektu graficznego.

W tym celu należy zdefiniować obiekt ze zmienioną procedurą Aktualizuj:

```
type Okno_Z_Odswiezaniem(Okno: access Podstawowe_Okno'Class) is
                    new Operacja_Graficzna with null record;
procedure Aktualizuj(M: in out Okno_Z_Odswiezaniem);
```

Treść procedury może być następująca:

```
procedure Aktualizuj(M: in out Okno_Z_Odswiezaniem) is
  -- Po prostu narysowanie obiektu
```



Rysunek 12.2: Lista operacji graficznych związanych z pewnym obiektem graficznym

```
begin
  Pokaz (M.okno.all); -- Skierowanie do właściwej procedury
end Update;
```

Teraz można połączyć zdefiniowany w ten sposób typ Okno_Z_Odswiezaniem z dowolnym typem opisującym okno w następujący sposób:

```
type Okno_Graficzne is new Moje_Okno with
  record
    OG: Okno_Z_Odswiezaniem (Okno_Graficzne'Access);
  end record;
```

gdzie składnik OG ma dyskryminant odnoszący się do typu zewnętrznego.

Tak utworzony element może być dołączony do listy obiektów graficznych (rysunek 12.3) i narysowany przez wywołanie procedury Notify.

Taka struktura naśladuje wewnętrzną reprezentację obiektów w obiektach wielokrotnie dziedziczących zadeklarowanych w innych językach.

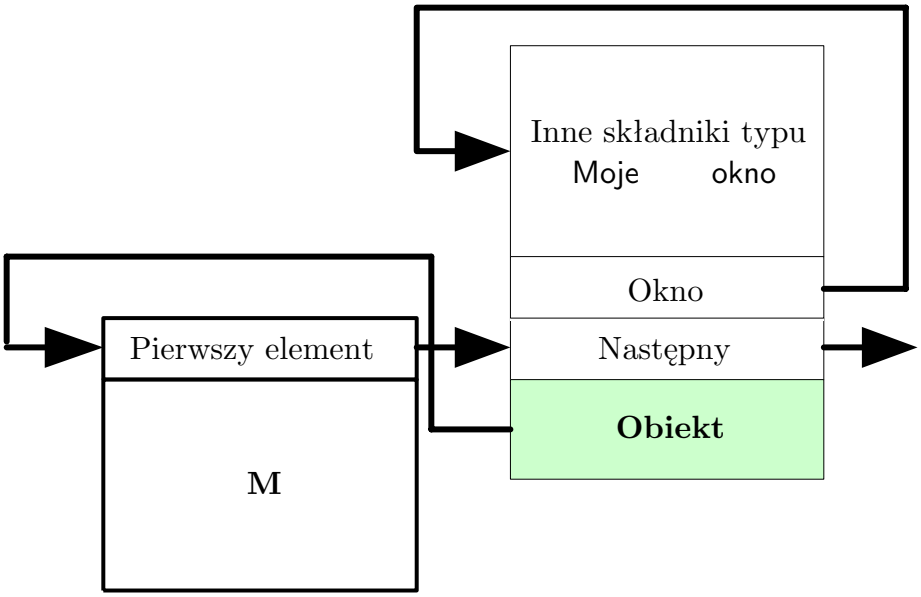
Więcej informacji na ten temat znajdzie Czytelnik i Czytelniczka w rozdziale 13.3.1

12.10 Przykłady programów obiektowych

12.10.1 Heterogeniczna lista dwukierunkowa

Wyjaśnienie czym jest „heterogeniczna lista dwukierunkowa” znajduje się na stronie 253. Do realizacji tej typowej techniki programistycznej bardzo dobrze nadają się opisywane w tym rozdziale obiekty.


```
1 package Podwojna_Lista is
2
3   type Wezel is tagged limited private;
4   type Wskaznik_Do_Wezla is access all Wezel'Class;
5
6   Niewłaściwy_Element : exception;
7
8   -- operacje wstawiania i usuwania z listy,
```



Rysunek 12.3: Obiekt Window_With_Graph w łańcuchu

```
9   -- na początek listy wskazuje zmienna "Poczatek"
10  procedure Dodaj(Element : Wskaznik_Do_Wezla;
11                  Początek : in out Wskaznik_Do_Wezla);
12  procedure Usun (Element : Wskaznik_Do_Wezla;
13                  Poczatek : in out Wskaznik_Do_Wezla);
14  -- funkcje do przeglądania listy
15  function Nastepny (Element: Wskaznik_Do_Wezla)
16                                  return Wskaznik_Do_Wezla;
17  function Poprzedni(Element: Wskaznik_Do_Wezla)
18                                  return Wskaznik_Do_Wezla;
19 private
20  type Wezel is tagged limited
21    record
22      Poprzedni: Wskaznik_Do_Wezla := null;
23      Nastepny : Wskaznik_Do_Wezla := null;
24    end record;
25 end Podwojna_Lista;
```

12

Operacje Dodaj, Usun, Nastepny i Poprzedni mogą być użyte dla każdego typu rozszerzającego typ Podwojna_Lista.Wezel.

Poniżej przedstawiono przykład użycia tak zdefiniowanej listy. Procedura Dodaj umieszcza nowy element na początku listy, procedura Usun usuwa z listy element. Procedury Nastepny i Poprzedni pozwalają na przemieszczenie się po liście.

W przypadku próby usunięcia elementu, który nie znajduje się na liście zgłoszony zostanie wyjątek Niewlasciwy_Element, podobnie jak w przypadku próby wstawienia elementu, który już jest na liście.

Przypuśćmy, że należy utworzyć tablicę związków pomiędzy różnymi typami danych. Jest to również dość popularna technika programistyczna. Jej celem

jest przyspieszenie wykonywania pewnych operacji. Np. jeżeli z każdym słowem związana jest pewna liczba całkowita to poszukiwanie pewnego słowa w słowniku można sprowadzić do porównywania liczb całkowitych, która to operacja jest wielokrotnie bardziej efektywna od operacji porównywania napisów. Operacja określania tej liczby całkowitej na podstawie napisu nazywa się *mieszaniem* (ang. *hashing*). Deklaracja odpowiedniego typu rozszerzonego i operacji na nim może być następująca.

```
1  with Podwojna_Lista;
2  generic
3    type Typ_Klucza is limited private;
4    with function "="(Lewy, Prawy: Typ_Klucza) return Boolean is <>;
5    with function Mieszanie(Klucz: Typ_Klucza) return Integer is <>;
6  package Związek is
7    type Typ_Elementu is new Podwojna_Lista.Wezel with
8      record
9        Klucz: Typ_Klucza;
10     end record;
11   type Wskaznik_Do_Elementu is new Podwojna_Lista.Wskaznik_Do_Wezla;
12
13   function Klucz(E: Wskaznik_Do_Elementu) return Typ_Klucza;
14
15   type Tablica_Zwiazkow(Rozmiar: Positive) is limited private;
16   -- "Rozmiar" określa rozmiar słownika
17   procedure Wprowadz(Tablica : in out Tablica_Związkow;
18                      Element : in Wskaznik_Do_Elementu);
19   function Szukaj(Tablica : in Tablica_Zwiazkow;
20                   Klucz : in Typ_Klucza) return Wskaznik_Do_Elementu;
21   -- inne operacje na tablicy związków ...
22 private
23   type Tablica_Wskazników_Do_Elementow is
24           array (Integer range <>) of Wskaznik_Do_Elementu;
25   type Tablica_Zwiazkow (Rozmiar: Positive) is
26     record
27       Zwiazki: Tablica_Wskaznikow_Do_Elementow (1 .. Rozmiar);
28     end record;
29 end Zwiazek;
```

Tablica_Zwiazkow to tablica mieszająca, w której każda wartość mieszająca jest związana z podwójną listą elementów. Elementy mogą być dowolnymi typami rozszerzającymi typ Typ_Elementu. Początek każdej listy jest typu Typ_Elementu, który pochodzi od typu Wskaznik_Do_Wezla, dlatego wszystkie podstawowe operacje na liście takie jak Usun, Dodaj itd. dotyczą także typu Wskaznik_Do_Elementu. Funkcja Klucz oblicza wartość pola Klucz obiektu danego jako parametr.

Powyższy obiekt można rozszerzyć tworząc np. listę symboli pozwalającą na rozbicie symboli na te, które odpowiadają typom, zmiennym, procedurom itd.

```
1  with Zwiazek;
2
3  package Pakiet_Tablic_Symboli is
4    type Identyfikator is access String;
5    -- Kluczem w tablicy symboli jest wskaźnik do napisu
6    -- co pozwala na korzystanie z identyfikatorow dowolnej długości
7    function Equal(Lewy, Prawy: Identyfikator) return Boolean;
```

```
8   function Mieszanie(Klucz: Identyfikator) return Integer;
9   -- konkretyzacja pakietu "Zwiazek" tworząca tablice symboli
10  package Zwiazek_Dot_Symboli is
11            new Zwiazek(Identyfikator, Equal, Mieszanie);
12  subtype Tablica_Symboli is Zwiazek_Dot_Symboli.Tablica_Zwiazkow;
13
14  -- definicja trzech rodzajów elementow w tablicy symboli
15  -- poprzez użycie rozszerzeń typów
16  type Symbole_Typow is new Zwiazek_Dot_Symboli.Typ_Elementu with
17    record
18      Kategoria : Kategorie_Typow;
19      Rozmiar : Natural;
20    end record;
21    type Wskaznik_Do_Typu is access Symbole_Typow;
22
23    type Symbole_Obiektow is new Zwiazek_Dot_Symboli.Typ_Elementu with
24      record
25        Typ_Obiektu : Wskaznik_Do_Typu;
26        Przesuniecie_Na_Stosie : Integer;
27      end record;
28
29    type Symbole_Funkcji is new Zwiazek_Dot_Symboli.Typ_Elementu with
30      record
31        Typ_Wyniku : Wskaznik_Do_Typu;
32        Parametry_Formalne : Tablica_Symboli(5);
33        -- bardzo mała tablica mieszająca
34        Zmienne_Lokalne : Tablica_Symboli (19);
35        -- większa tablica mieszająca
36        Tresc_Funkcji : Lista_Instrukcji;
37      end record;
38  end Pakiet_Tablic_Symboli;
```

Typ Tablica_Symboli jest utworzony poprzez konkretyzację pakietu ogólnego Zwiazek z kluczem, który jest wskaźnikiem do napisu. Zadeklarowane tu są trzy rozszerzenia typu Typ_Elementu, z których każdy ma odpowiadać innej klasie symboli

Treść pakietu ogólnego Zwiazek może być następująca:

```
1   package body Zwiazek is
2
3     procedure Wprowadz(Tablica : in out Tablica_Zwiazków;
4                        Element : Wskaznik_Do_Elementu) is
5                   -- Wprowadzanie nowego elementu do listy związków.
6       Indeks_Mieszajacy: constant Integer :=
7                 (Hash (Element.Klucz) mod Tablica.Rozmiar) + 1;
8       use Podwojna_Lista;
9     begin
10      -- Dodaj do kolekcji dot. właściwych związków
11      Dodaj (Element, Tablica.Zwiazki (Indeks_Mieszajacy));
12    end Enter;
13
14    function Klucz (E: Wskaznik_Do_Elementu) return Typ_Klucza is
15    begin
16      return Typ_Elementu(E.all).Klucz;
```

```
17    end Key;
18
19    function Szukaj(Tablica : Tablica_Zwiazkow;
20                    Klucz : Typ_Klucza) return Wskaznik_Do_Elementu is
21                 -- Wyszukiwanie odpowiedniego elementu w tablicy.
22       Indeks_Mieszajacy: constant Integer :=
23                 (Hash (Klucz) mod Tablica.Rozmiar) + 1;
24       Wskaznik: Wskaznik_Do_Elementu := Tablica.Zwiazki (Indeks_Mieszajacy);
25                  -- początek listy
26       use Podwojna_Lista;
27    begin
28       -- przeglądanie podwójnej listy w celu znalezienia takiego klucza
29       -- który odpowiada parametrowi "Klucz".
30       -- jeśli nie znajdzie, procedura oblicza wartość null.
31       while Wskaźnik /= null loop
32          if Klucz(Wskaznik).Klucz = Klucz then
33             return Wskaznik; -- "pasujący" element został znaleziony
34          end if;
35          Wskaznik := Nastepny (Wskaznik);
36       end loop;
37       return null; -- nie znaleziono odpowiedniego elementu
38    end Szukaj;
39
40 end Zwiazek;
```

Operacje Wprowadz i Szukaj są zaimplementowane przy pomocy operacji odziedziczonych po typie Wskaznik_Do_Elementu i Wskaznik_Do_Wezla.

12.10.2 Wielokrotne implementacje

Bardzo istotnym aspektem programów zorientowanych obiektowo jest możliwość dostarczania różnych implementacji pewnej abstrakcji jaką jest obiekt. Oczywiście nie jest to nic nowego, takie języki jak Modula-2 czy wcześniejsza wersja Ady pozwalały to robić już od dawna (Wiener i Sincovec 1984), jednak wybór implementacji musiał być dokonany na etapie konsolidowania programu. Jednakże czasami znacznie dogodniej jest dokonać wyboru implementacji już w czasie działania programu[8]. Konieczne jest przy tym, by zadbać o odpowiednią efektywność takiego połączenia.

Zastosowanie mechanizmów programowania obiektowego, wiążącego pewien typ danych z operacjami związanymi z tymi danymi, zapewnia możliwość takiego wyboru.

W programie obiektowym im bardziej podstawowy jest dany obiekt (tzn. im bliżej początku hierarchii się znajduje) tym bardziej ogólne operacje są w nim zaimplementowane. Każdy obiekt dziedziczący własności uszczegóławia konkretną implementację, przy czym różne takie obiekty mogą to robić, i zwykle robią, w inny sposób.

[8] Przykładem takiej operacji jest obiekt obsługujący obrazki, przy czym operacje zdefiniowane przez ten obiekt zależą od rodzaju obrazka (kolorowy, monochromatyczny, liczba kolorów itp.). Oczywiście informacja ta jest dostepna dopiero w czasie działania programu.

Poniżej pokazano przykład różnych implementacji tej samej abstrakcji jaką jest zbiór. Specyfikacja ogólnego pakietu Abstrakcyjny_Zbior realizującego najbardziej ogólne operacje na zbiorach może być następująca[9]:

```
1  subtype Element_Zbioru is Natural;
2  package Abstrakcyjny_Zbior is
3
4    type Zbior is abstract tagged private;
5    -- Oblicza zbiór pusty
6    function Pusty return Zbior is abstract;
7    -- Oblicza zbiór jednoelementowy
8    function Jednostka(Element: Element_Zbioru) return Zbior is abstract;
9    -- Suma dwu zbiorów
10   function Suma(Lewy, Right: Zbior) return Zbior is abstract;
11   -- Iloczyn dwu zbiorów
12   function Iloczyn(Lewy, Right: Zbior) return Zbior is abstract;
13   -- Usunięcie elementu ze zbioru
14   procedure Usun(Z : in out Zbior;
15                  Element: out Element_Zbioru) is abstract;
16   Za_Duzy_Element: exception;
17 private
18   type Zbior is abstract tagged null record;
19 end Abstrakcyjny_Zbior;
```

Tak zdefiniowany pakiet dostarcza całkowicie abstrakcyjnej definicji zbioru. Nawet typ Zbior jest zdefiniowany jako pusty abstrakcyjny rekord (chociaż jego postać jest ukryta), który **musi** być uzupełniony przez obiekty dziedziczące obiekt Zbior. Wszystkie podprogramy są również zdefiniowane jako abstrakcyjne, co oznacza, że nie mają one implementacji i nie mogą być bezpośrednio wywoływane, ale **muszą** być skonkretyzowane przez typy dziedziczące.

Jako przykład obiektu dziedziczącego wybrano typ Zbior_Bitow wraz z podprogramami zdefiniowanymi w pakiecie Zbior_Wektorow_Bitowych.

```
1  with Abstrakcyjny_Zbior;
2
3  package Zbior_Wektorów_Bitowych is
4
5    type Zbior_Bitow is new Abstrakcyjny_Zbior.Zbior with private;
6    -- Konkretne realizacje operacji abstrakcyjnych
7    function Pusty return Zbior_Bitow;
8    function Jednostka (Element: Element_Zbioru) return Zbior_Bitow;
9    function Suma (Lewy, Prawy: Zbior_Bitow) return Zbior_Bitow;
10   function Iloczyn (Lewy, Prawy: Zbior_Bitow) return Zbior_Bitow;
11   procedure Usun(Z : in out Zbior_Bitow;
12                  Element: out Element_Zbioru);
13 private
14   Rozmiar_Zbioru_Bitowego: constant := 64;
15   type Wektor_Bitow is
16     array (Element_Zbioru range 0 .. Rozmiar_Zbioru_Bitowego -1) of
17               Boolean;
18   pragma Pack(Wektor_Bitow);
19     -- oznacza to, że każdej wartości odpowiadać będzie jeden bit
```

[9] O spsobie tworzenia zbiorów w Adzie można przeczytać w rozdziale 11.6.

```
20    type Zbior_Bitow is new Abstrakcyjny_Zbior.Zbior with
21      record
22        Dane: Wektor_Bitow;
23      end record;
24  end Zbior_Wektorow_Bitowych;
25
26  package body Zbior_Wektorow_Bitowych is
27
28    function Pusty return Zbior_Bitow is
29    begin
30      return (Dane => (others => False));
31    end Pusty;
32
33    function Jednostka (Element: Element_Zbioru) return Zbior_Bitow is
34            S: Zbior_Bitow := Pusty;
35    begin
36      S.Dane (Element) := True;
37      return S;
38    end Jednostka;
39
40    function Suma (Lewy, Prawy: Zbior_Bitow) return Zbior_Bitow is
41    begin
42      return (Dane => Lewy.Dane or Prawy.Dane);
43    end Suma;
44
45    ...
46  end Zbior_Wektorow_Bitowych;
```

Inna implementacja może, zamiast na wektorze, operować na liście opisującej elementy, z których składa się dany zbiór (jest to odpowiednia konstrukcja dla bardzo rzadkich zbiorów). Korzystając z mechanizmów programowania obiektowego można zapewnić, że w jednym programie będą istniały obie postacie struktur opisujących zbiory. Konwersję pomiędzy nimi można zrealizować następująco:

```
1  procedure Konwersja(Z_Typu : in Zbior'Class;
2                      Do_Typu: out Zbior'Class) is
3    t : Zbior'Class := Z_Typu;
4    Elem : Element_Zbioru;
5  begin
6    -- budowanie zbioru docelowego, po jednym elemencie na raz
7    Do_Typu := Pusty;
8    while t /= Pusty loop
9      Usun (t, Elem);
10     Do_Typu := Suma (Do_Typu, Jednostka (Elem));
11   end loop;
12 end Konwersja;
```

Procedura ta dobiera odpowiednią operację zgodnie z rzeczywistym typem jej parametrów. Należy pamiętać, że wszystkie zmienne typu klasowego (takie jak t) muszą być zainicjowane, **ponieważ bez inicjacji nie są zgodne z żadnym rzeczywistym typem**. Operator porównania również zostanie dobrany do typu zmiennej, tak, że wyrażenie t /= Pusty używa operacji porównania dziedziczonej z obiektu Z_Typu. Ponadto, przypisanie jest także operacją dobieraną

wg typu zmiennej, co często uchodzi uwadze programistów. W przedstawionym przykładzie, jeżeli typ parametru Z_Typu jest elementem tworzącym listę, to zwykłe kopiowanie mogłoby tę listę uszkodzić. Dlatego należy tu użyć obiektów nadzorowanych (rozdział 12.7).

Ponadto możliwe jest takie napisanie pakietu obsługującego zbiory, który byłby pakietem ogólnym:

```
generic
  type Element_Zbioru is private;
package Abstrakcyjny_Zbior is ...
```

co pozwala na dodatkowe możliwości rozszerzeń (rozdział 13).

12.11 Iteratory

Jednym z częściej spotykanych wymagań podczas pisania programów jest konstrukcja umożliwiająca wykonanie pewnej operacji na każdym elemencie danego zbioru (listy). Jednym z podejść do tego problemu jest użycie dyskryminantów wskaźnikowych (rozdział 13.3.1), jednakże istnieje prostsza technika polegająca na użyciu rozszerzeń typów i polimorfizmu.

Początkowo rozważane będą przykłady nie korzystające z ogólnych jednostek programowych; skorzystamy z nich w dalszej części rozdziału

```
1  type Element is ...
2
3  package Zbiory is
4     type Zbior is limited private;
5     ... -- różne operacje na zbiorach
6     type Iterator is abstract tagged null record;
7
8     procedure Iteruj(S: Zbior; IC: Iterator'Class);
9     procedure Akcja(E: in out Element;
10                    I: in out Iterator) is abstract;
11 private
12    type Wezel;
13    type Wskaznik is access Wezel;
14    type Wezel is
15     record
16        E: Element;
17        Nastepny: Wskaznik;
18     end record;
19  type Zbior is new Wskaznik;
20 -- zaimplementowany jako pojedynczo połączona lista
21 end Zbiory;
22
23 package body Zbiory is
24    ... - Tresc roznych operacji na zbiorach
25
26    procedure Iteruj(S: Zbiór; IC: Iterator'Class) is
27       t : Wskaznik := Wskaznik (S);
28    begin
29       while t /= null loop
```

```
30          Akcja(t.E, IC); -- wykonanie akcji dobranej do typu
31          t := t.Nastepny;
32       end loop;
33    end Iteruj:
34
35 end Zbiory;
```

Takie rozwiązanie wprowadza abstrakcyjny obiekt Iterator wraz z podstawowym, abstrakcyjnym podprogramem Akcja. Procedura Iteruj przegląda zbiór (w tym przypadku zaimplementowany jako lista) i wywołuje odpowiednią dla danego typu procedurę Akcja. Można zatem powiedzieć, że głównym zadaniem procedury Iterator jest, oczywiście prócz przeglądania listy, identyfikacja typu zrealizowana przez dobór właściwej procedury Akcja. Poniższy, prosty przykład pokazuje sposób obliczania liczby elementów w zbiorze:

```
1  package Zbiory.Dodatkowe_Operacje is
2     function Licz(S: Zbiór) return Natural;
3  end Zbiory.Dodatkowe_Operacje;
4
5  package body Zbiory.Dodatkowe_Operacje is
6
7     type Iterator_Liczacy is new Iterator with
8        record
9           Wynik: Natural := 0;
10       end record;
11
12    procedure Akcja(E: in out Element;
13                    I: in out Iterator_Liczacy) is
14    begin
15       I.Wynik := I.Wynik + 1;
16    end Akcja;
17
18    function Licz(S: Zbior) return Natural is
19       I: Iterator_Liczacy;
20    begin
21       Iteruj (S, I);
22       return I.Wynik;
23    end Licz;
24 end Zbiory.Dodatkowe_Operacje;
```

Typ Iterator_Liczacy rozszerza abstrakcyjny typ Iterator, a specyficzna dla tego typu procedura Akcja wykonuje właściwe obliczanie. Wynik jest gromadzony w polu Wynik, co pozwala mu być widocznym w procedurze Akcja. Pole to jest inicjowane na zero, gdy Iterator_Liczacy jest deklarowany w treści funkcji Licz.

Ponadto, jeśli podstawowy pakiet Zbiory byłby pakietem ogólnym z typem Element jako formalnym parametrem, tak jak to zostało przedstawione w rozdziale 13, to pakiet potomny Zbiory.Dodatkowe_Operacje powinien także być pakietem ogólnym. W tym przypadku konieczne byłoby umieszczenie rozszerzenia typu i właściwej dla rozszerzonego typu operacji Akcja w prywatnej części pakietu Zbiory.Dodatkowe_Operacje z powodów wyjaśnionych dokładniej w rozdziale 13.1.1. Bardziej ogólne operacje mogą być napisane w podobny sposób. Każdy parametr i wynik operacji jest przesyłany jako składnik typu iteratora. Taką bardziej ogólna procedurą mogłoby być:

```
procedure Ogolna(S: Zbior; P: Parametry) is
  I: Ogolny_Iterator;
begin
  ... -- kopiowanie parametrów do iteratora
  Iteruj(S, I);
  ... -- kopiowanie wyników z iteratora do parametrów
end Ogolna;
```

a typ Ogolny_Iterator i odpowiadająca mu procedura Akcja mogła by mieć następującą formę:

```
type Ogolny_Iterator is new Iterator with
  record
    ... – odpowiednie składniki i parametry
  end record;

procedure Akcja(E: in out Element;
                I: in out Ogolny_Iterator) is
begin
  E := ...; -- robi coś z elementam korzystając z danych
            -- związanych z iteratorem
end Akcja;
```

Rozwiązanie korzystające z procedur polimorficznych przypomina rozwiązanie wykorzystujące wskaźniki do procedur. Jednakże ta analogia nie jest pełna:

```
procedure Iteruj(S : Zbior;
                 Akcja: access procedure(E: in out Element)) is
  t : Wskaznik := Wskaznik(S);
begin
  while t /= null loop
    Akcja(t.E);
    t := t.Nastepny;
  end loop;
end Iteruj;
```

skorzystanie z takiej funkcji może być następujące

```
function Licz(S: Zbior) return Natural is
  Wynik: Natural := 0;

  procedure Akcja_Liczenia(E: in out Element) is
    begin
      Wynik := wynik + 1;
    end Akcja_Liczenia;

  begin
    Iteruj(S, Akcja_Liczenia'Access);
    return Wynik;
 end Licz;
```

Niestety takie rozwiązanie jest niewłaściwe, ponieważ wskaźnik do procedury lokalnej Akcja_Liczenia jest niemożliwy (zob. rozdział 8.1.1), ale z drugiej strony procedura ta musi być lokalna po to, by można w niej było manipulować zmienną Wynik. Zmienna ta nie powinna być zmienną globalną, ponieważ sprawiałaby kłopoty w programach wielozadaniowych (rozdział 14.5). Poza tym, nie

jest właściwym podejściem tworzenie zmiennej istniejącej przez cały czas życia programu (a taką jest każda zmienna globalna), mimo, że taka zmienna jest używana stosunkowo rzadko.

12.12 Dobór procedur do typu parametru

W celu rozproszenia wszelkich wątpliwości co do tego, kiedy następuje dynamiczny (tj. w czasie wykonania programu) dobór wywoływanej procedury, a kiedy statyczny (tj. na etapie kompilacji) w niniejszym rozdziale zostanie ten problem poddany głębszej dyskusji.

Każdy obiekt posiada *znacznik* (ang. *tag*), pozwalający określić, rzeczywisty typ zmiennej obiektowej. Jeżeli pewna zmienna typu Pewien_Typ'Class jest parametrem procedury posiadającej wiele implementacji, z których każda dotyczy innego rozszerzenia pewnego obiektu, to program musi w miejscu wywołania dobrać wywołanie do rzeczywistego typu zmiennej określonego przez znacznik.

Dobór ten jest dokonywany w taki sposób, że wszystkie parametry podprogramu (i, ewentualnie, wynik funkcji) muszą być zgodne z deklaracją nagłówka (czyli odnosić się do pewnego konkretnej, ściśle określonej metody obiektu). Jeśli nie uda się dobrać takiego podprogramu, to zgłaszany jest wyjątek CONSTRAINT_ERROR. W Adzie niedozwolone jest takie użycie procedury polimorficznej (metody wirtualnej), w której część typów parametrów jest określona statycznie, a część dynamicznie np. (dla obiektów opisanych w poprzednim podrozdziale):

```
S: Zbior_Bitow := ...
T: Zbior'Class := ...
...
S := Suma(S, T); -- nielegalne
```

chociaż w czasie wykonania programu takie wywołanie mogłoby się udać, jeżeli nagłówek związany ze zmienną T miałby wartość Zbior_Bitow'Tag. Oczywiście można w takim wypadku użyć następującego wywołania funkcji Suma:

```
S := Suma (S, Zbior_Bitow (T));
```

przy czym konwersja z typu T na typ Zbior_Bitow może być również przyczyną błędu wykonania programu w przypadku niezgodności typów. Właściwy typ zmiennej T musi być typem dziedziczącym własności po typie Zbior_Bitow, czyli nagłówki tego typu nie muszą odpowiadać typowi Zbior_Bitow.

Czasami jednak nagłówek (czyli dokładny typ) jest niemożliwy do określenia na podstawie tylko parametrów podprogramu. Np. w procedurze Konwersja wprowadzonej w poprzednim rozdziale pojawia się instrukcja:

```
Do_Typu := Pusty;
```

Bezparametrowa funkcja Pusty oblicza wynik nieokreślonego typu (ściślej – istnieją różne wersje funkcji Pusty, z których każda związana jest z innym rozszerzeniem typu Zbior). O typie tym wiadomo tylko tyle, że jest on typem rozszerzającym typ Zbior. Dlatego też dobór odpowiedniej funkcji odbędzie się na podstawie typu wyniku tej funkcji, czyli na podstawie typu zmiennej Do_Typu. Podobnie instrukcja:

```
Do_Typu := Suma (Do_Typu, Jednostka (Elem));
```

powoduje dobranie najpierw właściwej procedury Suma na podstawie typu zmiennej Do_Typu. Po wybraniu właściwej procedury Suma, na podstawie jej parametrów ustalić można odpowiednią wersję procedury Jednostka.

12.13 Ćwiczenia

◁ Ćwiczenie 12.1 ▷▷▷ Stwórz dwie klasy A i B z domyślnymi konstruktorami, które siebie przedstawiają. Zbuduj na bazie klasy A klasę C i stwórz składową klasy B wewnatrz klasy C. Stwórz obiekt klasy C i zobacz jak działa.

◁ Ćwiczenie 12.2 ▷▷▷ Zmodyfikuj poprzednie ćwiczenie tak, aby klasy A i B posiadały konstruktory z parametrami. Dopisz konstruktor dla klasy C i wykonaj całą inicjację wewnątrz konstruktora klasy C

◁ Ćwiczenie 12.3 ▷▷▷ Stwórz hierarchie dziedziczenia dla gryzoni: klasy Mysz, Chomik itd. W klasie bazowej umieść metody wspólne dla wszystkich gryzoni, a następnie nadpisz je realizując różnorodne zachowanie sie klas potomnych. Stwórz tablicę gryzoni, przy czym wywołuj metody klasy bazowej, obserwując co sie dzieje.

◁ Ćwiczenie 12.4 ▷▷▷ Zmodyfikuj poprzednie ćwiczenie czyniąc metody klasy gryzoń abstrakcyjnymi, gdy to tylko mozliwe.

◁ Ćwiczenie 12.5 ▷▷▷ Zmodyfikuj poprzednie ćwiczenie tak, aby demonstrowało kolejność inicjacji klas bazowych i pochodnych. Następnie dodaj obiekty składowe zarówno do klasy bazowej, jak i pochodnej, po czym sprawdź, w jakiej kolejności następuje inicjacja w trakcie konstrukcji.

◁ Ćwiczenie 12.6 ▷▷▷ Stwórz klase abstrakcyjną bez żadnych metod abstrakcyjnych i sprawdź, że nie możesz utworzyć obiektów tej klasy.

◁ Ćwiczenie 12.7 ▷▷▷ Stwórz klasę nieabstrakcyjną i utwórz procedurę abstrakcyjną mającą jako parametr klasowy tę klasę. Sprawdź reakcję kompilatora.

◁ Ćwiczenie 12.8 ▷▷▷ Stwórz trójpoziomową hierarchię dziedziczenia na bazie pakietu Ada.Finalization. Sprawdź, że poprawnie działa „niszczenie” obiektów tej klasy.

◁ Ćwiczenie 12.9 ▷▷▷ Stwórz klasę bazową z dwiema metodami. W pierwszej wywołaj drugą z nich. Wywiedź przez dziedziczenie klasę pochodną ze stworzonej klasy bazowej i nadpisz drugą z metod. Stwórz obiekt klasy potomnej i wywołaj dla niego pierwszą z metod, a nastepnie podstaw go do zmiennej klasy

bazowej i wywołaj pierwszą z metod dla zmiennej klasy bazowej. Wytłumacz co się dzieje.

◃ Ćwiczenie 12.10 ▹▹▹ Stwórz klasę bazową z abstrakcyjna metodą Put, nadpisana w klasie pochodnej. Wersja nadpisana wypisuje wartość zmiennej całkowitej zdefiniowanej w klasie potomnej. W miejscu definiowania tej zmiennej nadaj jej wartość. W konstruktorze klasy bazowej wywołaj tę metodę. Stwórz obiekt typu pochodnego i wywołaj metodę Put. Wytłumacz rezultat.

Rozdział 13

Ogólne jednostki programowe

Ogólne jednostki programowe są zwykle w polskojęzycznych publikacjach na temat języka Ada nazywane jednostkami rodzajowymi. Wydaje sie nam jednak, że tłumaczenie angielskiego *generic* jako jednostki *ogólne* znacznie lepiej oddaje ich sens niż, w gruncie rzeczy mocno nieokreślone słowo *rodzajowe*.

Chociaż tzw. „jednostką" ogólną (rodzajową) może być podprogram (tj. procedura lub funkcja) lub pakiet, to w niniejszej książce założymy, że jednostki ogólne będą reprezentowane przez pakiety ogólne. Warto tu jednak podać przykład stosunkowo często wykorzystywanej (głównie przy współpracy z bibliotekami napisanymi w języku C, który nie jest tak „napastliwy" przy sprawdzaniu poprawności jak Ada) funkcji ogólnej Ada.Unchecked_Conversion, zamieniającej dowolny typ na dowolny inny, nawet w przypadku, w którym wielkości zmiennych są całkowicie różne, zadowalając się tylko ostrzeżeniem o zaistnieniu takiej sytuacji.

Bardzo użytecznym mechanizmem pozwalającym na pisanie fragmentów kodu w taki sposób, by jak najłatwiej byłoby z nich korzystać są tzw. *makroinstrukcje* lub po prostu *makra*. Są to, w pewnym uproszczeniu, fragmenty tekstu, których nazwa w programie jest zamieniana na ich treść. Najprostsza wersja takiego makra może być makro min wyznaczające minimalną wartość z dwu liczb w postaci pseudoprogramu:

```
macro min(a, b)
    if a > b then
        return b;
    else
        return a;
    end if;
end min;
```

Wywołanie tak zdefiniowanego elementu programu jest takie same, niezależnie od tego czy a i b są zmiennym typu całkowitoliczbowego, znakowego, czy jakiegokolwiek innego, dla którego zdefiniowany jest operator „>". Niestety nic nie stoi też na przeszkodzie by jedna z tych zmiennych była napisem a druga

zmienną logiczną. Oczywiście kompilator odrzuci taką *konkretyzację* tego konkretnego makra, ale można sobie wyobrazić sytuację, w której błąd zostanie wykryty w innej części kodu, skądinąd poprawnej (każdy, kto używał makr w języku C musi się zgodzić z powyższym stwierdzeniem).

Dlatego też, zamiast makr, wprowadzono tzw. wzorce zwane w Adzie ogólnymi jednostkami programowymi[1] (mogą to być pakiety, funkcje lub procedury). Główną różnicą pomiędzy tymi konstrukcjami są ograniczenia związane z częściową typizacją owych ogólnych jednostek (czyli wzorców), a także sprawdzanie poprawności składniowej (w makroinstrukcji nikt nie żąda poprawności składniowej, jest to po prostu zastąpienie jednego tekstu przez inny).

W pewnym uproszczeniu można powiedzieć, że tak jak parametrami formalnymi podprogramów są stałe i zmienne, tak parametrami formalnymi pakietów ogólnych są także typy, procedury, pakiety itp.

Informacje zawarte w niniejszym rozdziale uzupełniają się wzajemnie z informacjami dotyczącymi programowania obiektowego (rozdział 12), w szczególności dotyczącymi wielokrotnego dziedziczenia (rozdział 12.8).

13.1 Podstawy

13.1.1 Parametry ogólne

Ponieważ przetwarzanie ogólnych jednostek programowych odbywa się w czasie kompilacji programu, a nie w czasie jego wykonania, to możliwe jest tylko częściowe (niepełne) określenie ogólnych typów danych, co pociąga za sobą częściowe określenie podprogramów zawartych w pewnej jednostce.

W Adzie stosowane są następujące ogólne (czyli niepełne) określenia typów:

limited private	klasa wszystkich typów
private	klasa wszystkich typów nieograniczonych
tagged limited private	klasa wszystkich typów obiektowych
tagged private	klasa wszystkich nieograniczonych typów obiektowych

Ponieważ określenia takie jak „klasa wszystkich typów” pozwalają by parametrem formalnym był każdy typ, to operacje na zmiennych tego typu są takie same jak operacje na każdych innych zmiennych typu ograniczonego (rozdział 9.3) tj. mogą być używane tylko jako parametry odpowiednich funkcji. Jeżeli dany typ nie jest typem ograniczonym, to można używać operacji podstawienia i porównania. To samo dotyczy typów obiektowych, z tym wyjątkiem, że można używać w tym wypadku atrybutu 'Class. Jeżeli obiektowy typ ogólny jest typem abstrakcyjnym (np. abstract tagged limited private), to nie można oczywiście tworzyć zmiennych tego typu (rozdział 12.5).

[1] W języku C++ (Stroustrup 1991) od pewnego czasu również można korzystać ze wzorców (ang. *templates*)

Czasami warto jednak pewien typ określić bardziej ściśle, po to, by można było na zmiennych typu ogólnego wykonywać pewne operacje. Poniżej opisano sposoby deklaracji różnego rodzaju klas typów ogólnych.

Typy skalarne

(<>)	typ dyskretny tj. całkowitoliczbowy lub wyliczeniowy (rozdział 3.4).
range <>	typ całkowitoliczbowy (rozdział 3.7).
mod <>	typ modularny (rozdział 11.5).
delta <>	typ stałopozycyjny (rozdział 11.3).
digits <>	typ zmiennopozycyjny (rozdział 3.8).
delta <> **digits** <>	dziesiętny typ stałopozycyjny (rozdział 11.3.1).

Mimo tego, że typy modularne są w zasadzie typami całkowitymi, to ze względu na fundamentalne różnice, które dzielą te typy, (rozdział 11.5) nie są one zgodne z typami całkowitymi. Podobnie dziesiętny typ stałopozycyjny (rozdział 11.3.1) jest niezgodny ze „zwykłym" typem stałopozycyjnym (rozdział 11.3).

Typy tablicowe

Deklaracja ogólnych typów tablicowych jest bardzo zbliżona do deklaracji „zwyczajnych" typów tablicowych, przy czym jedyną formą deklaracji dyskretnego podtypu służącego do opisu rozmiaru tablicy jest deklaracja typu dyskretnego, który musi być również zadeklarowany w części deklaracyjnej pakietu ogólnego.

Formalny i aktualny typ tablicowy muszą spełniać następujące wymagania:

1. Każdy wymiar w obu tych typach musi sobie odpowiadać, tzn. podtypy używane jako indeksy (w przypadku wymiaru tablicy o nieokreślonym rozmiarze) lub odpowiednie zakresy (w przypadku wymiaru tablicy o określonym rozmiarze) muszą sobie odpowiadać.
2. Typy elementów tablicy także muszą sobie odpowiadać.

Przykład deklaracji pewnego pakietu ogólnego:

```
1 generic
2     type Element is private;
3     type Indeks is (<>);
4     type Wector is array (Indeks range <>) of Element;
5     type Tablica is array (Indeks) of Element;
6 package P is
7     ...
8 end P;
```

a konkretyzacja takiej ogólnej jednostki programowej mogłaby być następująca:

```
type Mix is array (Kolor range <>) of Boolean;
type Opcja is array (Kolor) of Boolean;
-- Typ aktualny Mix "pasuje" do typu Wector, a także
-- typ aktualny Opcja "pasuje" do typu Tablica
package R is new P(Element => Boolean,
                   Indeks => Kolor,
                   Wector => Mix,
                   Tablica => Opcja);
```

Warto zauważyć, że typ aktualny Opcja „nie pasuje" do typu Wector, podobnie jak typ aktualny Mix „nie pasuje" do typu Tablica, ze względu na to, że typ Wector ma nieokreślony rozmiar, natomiast typ Tablica ma rozmiar określony ściśle zdefiniowanym typem Indeks.

Konkretyzacja może być zrealizowana tak, jak to zostało przedstawione w powyższym przykładzie, a może też mieć nieco mniej czytelną, ale prostszą formę:

```
package R is new P(Boolean, Kolor, Mix, Opcja);
```

Typy wskaźnikowe

Każdy aktualny typ wskaźnikowy „pasuje" do każdego formalnego typu wskaźnikowego, co oczywiście nie oznacza, że użycie jednostek ogólnych powoduje jakąkolwiek utratę bezpieczeństwa programu. Jeżeli formalny wskaźnikowy typ ogólny zadeklarowany jest ze specyfikacją constant (rozdział 8), to aktualny typ wskaźnika również musi być wskaźnikiem zadeklarowanym ze specyfikacją constant. Podobna sytuacja dotyczy też specyfikacji all (rozdział 8). Deklaracja ogólnego pakietu ze wskaźnikowymi parametrami formalnymi może wyglądać następująco:

```
generic
   type Wezel is private;
   type Polaczenie is access Wezel;
   package P is
      ...
   end P;
```

a jej konkretyzacja

```
type Samochod;
type Nazwa_Samochodu is access Samochod;
type Samochod is
   record
      Poprzednik, Nastepnik : Nazwa_Samochodu;
      Numer_Rejestracyjny : String(1..7);
      Wlasciciel : Opis_Osoby;
   end record;

package R is new P(Wezel => Samochod,
                   Polaczenie => Nazwa_Samochodu);
```

Wskaźniki do podprogramów

Podprogramami, do których wskaźniki mogą być przekazywane do ogólnych jednostek programowych mogą być funkcje lub procedury, natomiast **nie mogą** to być bariery (rozdział 14.5). Poniżej zademonstrowano proste przykłady deklaracji ogólnych podprogramów:

```
1 with function "+"(X, Y : Element) return Element is <>;
2 with function Image(X : Typ_Wyliczeniowy) return String
3                       is Typ_Wyliczeniowy'Image;
4 with procedure Aktualizuj is Standardowa_Deklaracja;
```

Jeśli dana jest sekcja ogólna w następującej postaci:

```
1 generic
2    with procedure Akcja(X : in Element);
3    procedure Iteruj(Sekwencja : in Sekwencja_Elementow);
```

i dana jest procedura

```
procedure Wstaw_Element(X : in Element);
```

możliwa jest następująca konkretyzacja

```
procedure Wstaw_Na_Liste is new Iteruj(Akcja => Wstaw_Element);
```

Typy obiektowe

Istnieją dwa rodzaje deklaracji ogólnych typów obiektowych:

```
type T is tagged private;
```

który jest odpowiednikiem całkiem dowolnego typu obiektowego i

```
type T is new S with private;
```

który określa, że typ T jest dowolnym typem dziedziczącym własności po typie S (oczywiście typ T może być tożsamy z typem S).

Ta druga forma jest bardzo istotna, ponieważ stanowi formę wielokrotnego dziedziczenia (rozdział 12.8). Warto tu zwrócić uwagę na to, że **w treści** ogólnej jednostki programowej nie można tworzyć nowych typów pochodnych typu T, ponieważ jest prawdopodobne, że typ T ma już inne typy pochodne. Nic natomiast nie stoi na przeszkodzie by zdefiniować rozszerzenie typu T w części deklaracyjnej pakietu ogólnego.

Ogólna deklaracja abstrakcyjnego typu ogólnego nie oznacza, że typ aktualny musi być również typem abstrakcyjnym. Oznacza to tylko tyle, że nie można zadeklarować obiektu tego typu, tak jak to jest w każdym innym przypadku użycia typu abstrakcyjnego (rozdział 12.5). Nieco bardziej złożony jest problem wywołań podprogramów abstrakcyjnych. W Adzie nie można wywołać podprogramu abstrakcyjnego, dlatego, o ile w momencie konkretyzacji pewnego pakietu ogólnego znana jest nieabstrakcyjna implementacja pewnej metody to można ją wywołać, w przeciwnym razie – nie.

```
generic
   type Rodzic is abstract new Limited_Controlled with private;
package P is
   type T is new Rodzic with private;
   ...
private
   type T is new Rodzic with
      record
         -- dodatkowe skladniki
      end record;

   procedure Finalize(Object: in out T);
end P;
```

chociaż typ Limited_Controlled (rozdział 12.7) jest typem abstrakcyjnym, to operacje na nim zdefiniowane takie jak Finalize nie są abstrakcyjne i dlatego możliwe jest wywołanie tej procedury w treści zadania. Ważne jest by zauważyć, że typ aktualny nie może zamienić procedury Finalize na wersję abstrakcyjną (Intermetrics Inc. 1995*b*, §3.9.3). Dlatego następująca deklaracja jest nielegalna

```
type Obiekt is abstract new Limited_Controlled with null record;
procedure Finalize(O: in out Obiekt) is abstract;
...
package Q is new P(Rodzic => Obiekt); -- nielegalna konkretyzacja
```

Programowanie obiektowe w połączeniu z ogólnymi jednostkami programowymi daje wiele nowych możliwości. Ogólne jednostki programowe mogą być parametryzowane poprzez obiekty pozwalając rozbudowywać obiekty będące parametrami pewnej ogólnej jednostki programowej. W przykładzie typów związanych z kontami (strona 239), typ Konto, użyty do konkretyzacji przedstawionego niżej pakietu Zbior_Kont (parametr Typ_Konta) będzie „pasował" do każdego typu dziedziczonego z typu Historia_Konta. We wzorcu dostępne są podstawowe operacje zdefiniowane dla typu Historia_Konta.

```
generic
   type Typ_Konta(<>) is new Historia_Konta with private;
package Zbior_Kont is
   procedure Dodaj_Nowe_Konto(A: in Typ_Konta);
   procedure Usun_Konto(A: in Typ_Konta);
   function Stan_Konta return Pieniadze;
   ... -- inne operacje
end Zbior_Kont;
```


Ogólny pakiet programowy może być skonkretyzowany z dowolnym typem dziedziczącym wartości z typu Historia_Konta. W tym przypadku pakiet tworzy homogeniczną listę kont. Można jednak skonkretyzować ten pakiet z typem klasowym np. Historia_Konta'Class, co pozwala na stworzenie heterogenicznej listy kont. Notacja (<>) określa, że aktualny typ konta może mieć dowolną liczbę dyskryminantów lub może być typem klasowym (tzn. może być nie w pełni zdefiniowany).

13.1.2 Deklaracja ogólnej jednostki programowej

Deklarację jednostki ogólnej tworzy się poprzedzając specyfikację podprogramu lub pakietu słowem kluczowym generic. Następnie pojawia się sekcja parametrów ogólnych określająca te parametry w sposób opisany w rozdziale 13.1.1. Po sekcji parametrów ogólnych znajduje się normalna specyfikacja pakietu lub procedury, w której można oczywiście używać uprzednio zdefiniowanych parametrów ogólnych np.:

```
1  generic
2    type Liczba is range <>;
3  package Wejscie_Wyjscie_Dla_Liczb is
4    Standardowa_Szerokosc : Pole := Liczba'Width;
5    Standardowa_Podstawa : Podstawa_Liczbowa := 10;
6
7    procedure Czytaj(Element : out Liczba;
8                     Szerokosc : in Pole := 0);
9
10   procedure Pisz
11       (Element : in Liczba;
12        Szerokosc : in Pole := Standardowa_Szerokosc;
13        Podstawa: in Podstawa_Liczbowa := Standardowa_Podstawa);
14
15 end Wejscie_Wyjscie_Dla_Liczb;
```

Pakiet ten zawiera dwie procedury Czytaj i Pisz operujące na zmiennej Liczba, która jest pewnym typem całkowitym. Typ ten jednak nie jest ani typem Integer, ani Positive, ani żadnym innym konkretnym typem. Zarówno jednak typ Integer, jak i Positive jest zgodny z typem Liczba. Można dokonać konkretyzacji takiej jednostki poprzez deklarację:

```
package Wejscie_Wyjscie_Dla_Liczb_Naturalnych is new
                                    Wejscie_Wyjscie_Dla_Liczb (Natural);
```

13.1.3 Treść ogólnej jednostki programowej

Treść ogólnej jednostki programowej prawie nie różni się od treści odpowiedniej konkretnej (nieogólnej) procedury lub pakietu. Można deklarować i używać zmiennych typu ogólnego tak, że treść napisanych jednostek nie różni się w zasadzie od treści tych samych jednostek napisanych w sposób nieogólny. Jedną z różnic, na którą należy zwrócić uwagę, jest zakaz pobierania w treści jednostki ogólnej wskaźnika do podprogramu spoza tej treści. Szczegółowa dyskusja tego problemu znajduje się w (Intermetrics Inc. 1995*a*, Intermetrics Inc. 1995*b*), Inne różnice opisane są w rozdziale 13.1.1.

W poniższych przykładach zaznaczono, które typy są typami ogólnymi:

```
1  procedure Wymien (U, V : in out Element) is
2    T : Element; -- Element jest typem ogólnym
3  begin
4    T := U;
5    U := V;
6    V := T;
```

```
7  end Wymień;
8
9  function Podnoszenie_Do_Kwadratu(X : Pewna_Liczba) return Pewna_Liczba is
10 -- Pewna_Liczba jest parametrem ogólnym
11 begin
12   return X*X; -- operator "*" jest operatorem formalnym
13 end Podnoszenie_Do_Kwadratu;
14
15 package body Wektory is
16
17   function Suma(A, B : Wector) return Wector is
18     -- Wector jest parametrem ogólnym
19     Wynik : Wector(A'Range);
20     Przesuniecie : constant Integer := B'First - A'First;
21   begin
22     if A'Length /= B'Length then
23       raise Niewlasciwa_Dlugosc;
24     end if;
25     for N in A'Range loop
26       wynik(N) := Suma(A(N), B(N + Przesuniecie));
27       -- funkcja Suma jest funkcją formalną
28     end loop;
29     return wynik;
30   end Sum;
31
32   function Sigma(A : Wector) return Element is
33   -- Element jest parametrem ogólnym
34     Suma_Elementow : Element := A(A'First);
35   begin
36     for N in A'First + 1 .. A'Last loop
37       Suma_Elementow := Suma(Suma_Elementow, A(N));
38       -- funkcja Suma jest funkcją formalną
39     end loop;
40     return Suma_Elementow;
41   end Sigma;
42
43 end Wektory;
```

13.1.4 Użycie ogólnej jednostki programowej

Aby użyć zdefiniowanego w poprzednim punkcie pakietu ogólnego należy użyć deklaracji (zakładając, że pakiet powyżej zdefiniowany jest widoczny) zwanej *konkretyzacją pakietu ogólnego*:

```
package Wejscie_Wyjscie_Dla_Liczb_Dodatnich is new
        Wejscie_Wyjscie_Dla_Liczb (Positive);
```

Od tej chwili wywołanie Wejscie_Wyjscie_Dla_Liczb_Dodatnich.Pisz_dodatnia niczym nie różni się od „normalnego” wywołania procedury. Jeśli to konieczne, to nastąpi sprawdzenie czy zm_dodatnia jest rzeczywiści zmienną dodatnią, a jeśli tak nie jest – to nastąpi zgłoszenie wyjątku CONSTRAINT_ERROR.

Podejście polegające na tworzeniu jednostek ogólnych pozwala na łatwiejsze tworzenie bardzo ogólnych bibliotek (tak ogólnych, że niemożliwe jest ich napisanie w zwykły sposób) bez ryzyka związanego z zaniechaniem ścisłej typizacji. Oczywiście, przy niemałej dozie dobrej woli, „zaciskając zęby", można utworzyć taki konkretny podprogram, który posiadałby odpowiednią liczbę parametrów pozwalających „doprecyzować" to, co chcielibyśmy uzyskać. Jednakże takie rozwiązanie ani nie jest eleganckie, ani efektywne, ani czytelne, ani praktyczne, ani nawet proste.

Inne przykłady konkretyzacji ogólnych jednostek programowych:

```
procedure Zamien is new Wymien (Element => Integer);
procedure Zamien is new Wymien (Character); -- "Zamien" jest przeciążone
function Kwadrat is new Podnoszenie_Do_Kwadratu(Integer);
         -- "*" dotyczący typu Integer
function Kwadrat is new
         Podnoszenie_Do_Kwadratu(Pewna_Liczba => Macierz,
         "*" => Iloczyn_Macierzy);
function Kwadrat is
         new Podnoszenie_Do_Kwadratu( Macierz, Iloczyn_Macierzy);
    -- równoważne poprzedniej konkretyzacji.
package Wektory_Calkowitoliczbowe is new Wektory(Integer, Tablica, "+");
```

Użycie skonkretyzowanej ogólnej jednostki programowej

```
Zamien(A, B);
A := Kwadrat(A);
T : Tablica(1 .. 5) := (10, 20, 30, 40, 50);
N : Integer := Wektory_Calkowitoliczbowe.Sigma(T);
use Wektory_Calkowitoliczbowe;
M : Integer := Sigma(T);
```

13.2 Parametry pakietowe

Najbardziej chyba złożonym parametrem formalnym jednostki ogólnej jest... inny pakiet. Formalny pakietowy parametr jednostki ogólnej jest zgodny z dowolną konkretyzacją określonego pakietu ogólnego.

Stosowanie ogólnych pakietów formalnych jest cennym mechanizmem szczególnie w dwóch okolicznościach. Po pierwsze, jednostki ogólne definiują dodatkowe operacje, wprowadzają nowy poziom abstrakcji. Taka warstwowa struktura jest szczególnie cenna, jeżeli wszystkie typy i operacje zdefiniowane w pewnym pakiecie ogólnym muszą być importowane do innego pakietu ogólnego. Innymi słowy, ogólne parametry formalne pozwalają pakietom ogólnym być parametryzowanym przez inne pakiety ogólne. W szczególności konstrukcja taka pozwala rozszerzyć abstrakcję opisaną w pewnym pakiecie bez konieczności wyliczania wszystkich operacji z pierwszej części formalnej jednostki w drugiej. Stanowi to logiczny odpowiednik operacji rozszerzania pakietów. Prostą ilustracją może być pakiet Ogolne_Wektory_Zespolone.

```
1 generic
2    type Typ_Zmiennoprzecinkowy is digits <>;
3 package Ogolne_Liczby_Zespolone is
```

```
4
5    type Zespolone is private;
6
7    function "+" (Lewy, Prawy: Zespolone) return Zespolone;
8    function "-" (Lewy, Prawy: Zespolone) return Zespolone;
9    -- itd.
10 end Ogolne_Liczby_Zespolone;
11
12 generic
13    type Typ_Zmiennoprzecinkowy is digits <>;
14    type Zespolone is private;
15
16    with function "+" (Lewy, Prawy: Zespolone) return Zespolone is <>;
17    with function "-" (Lewy, Prawy: Zespolone) return Zespolone is <>;
18    -- itd.
19 package Ogolne_Wektory_Zespolone is
20    ...
21 end Ogolne_Wektory_Zespolone;
```

Pakiety te można skonkretyzować w następujący sposób:

```
1  package Liczby_Zespolone_Podwojnej_Precyzji is
2     new Ogolne_Liczby_Zespolone(Long_Float);
3
4  use Liczby_Zespolone_Podwojnej_Precyzji;
5
6  package Wektory_Zespolone_Podwojnej_Precyzji is
7     new Ogolne_Wektory_Zespolone(Long_Float, Zespolone);
```

Czasami może zaistnieć potrzeba zastosowania kilku pakietowych parametrów formalnych. Prowadzi to do zastosowania następującej notacji:

```
with package P is new Q(<>);
```

co oznacza, że aktualny parametr odpowiadający P może być dowolnym uzyskanym poprzez konkretyzację Q w następujący sposób:

```
with package R is new Q(P1, P2, ...);
```

Deklaracja ta stwierdza, że aktualny pakiet odpowiadający R musi być konkretyzowany z podanymi parametrami.

Wracając do przykładu z liczbami zespolonymi, można, korzystając z opisanego mechanizmu, napisać pakiet, który implementuje standardowe operacje matematyczne na liczbach zespolonych, który ma dwa pakiety jako parametry: Pierwszy z nich definiuje operacje na liczbach zespolonych, a drugi z nich jest standardowym pakietem Generic_Elementary_Functions, dostarczającym elementarnych funkcji matematycznych operujących na liczbach rzeczywistych.

13

```
1  with Ogolne_Liczby_Zespolone;
2  with Generic_Elementary_Functions;
3
4  generic
5     with package Liczby_Zespolone is
6        new Ogolne_Liczby_Zespolone (<>);
7     with package Funkcje_Elementarne is
```

```
8         new Generic_Elementary_Functions
9                   (Liczby_Zespolone.Typ_Zmiennoprzecinkowy);
10    package Ogolne_Funkcje_Zespolone is
11       use Liczby_Zespolone;
12
13    function Sqrt(X: Zespolone) return Zespolone;
14    function Log (X: Zespolone) return Zespolone;
15    function Exp (X: Zespolone) return Zespolone;
16    function Sin (X: Zespolone) return Zespolone;
17    function Cos (X: Zespolone) return Zespolone;
18
19 end Ogolne_Funkcje_Zespolone;
```

Pakiety aktualne muszą być konkretyzacją odpowiednio pakietów Ogolne_Liczby_Zespolone i Generic_Elementary_Functions. Należy zauważyć, że używane są obie formy pakietów formalnych. Dozwolona jest dowolna konkretyzacja pakietu Ogolne_Liczby_Zespolone, ale tylko taka konkretyzacja pakietu Generic_Elementary_Functions, której aktualnym parametrem jest typ Liczby_Zespolone.Typ_Zmiennoprzecinkowy. Służy to zapewnieniu takiej sytuacji, w której oba pakiety są skonkretyzowane z tym samym typem zmiennoprzecinkowym. Parametr formalny utworzony w wyniku pierwszej konkretyzacji używany jest w drugiej. Parametry formalne są dostępne w taki sposób tylko wtedy, kiedy są użyte w standardowej formie (<>). Ostateczna konkretyzacja mogłaby być następująca:

```
1  package Liczby_Zespolone_Podwojnej_Precyzji is
2     new Ogolne_Liczby_Zespolone(Long_Float);
3
4  package Funkcje_Elementarne_Podwojnej_Precyzji is
5     new Generic_Elementary_Functions(Long_Float);
6
7  package Funkcje_Zespolone_Podwojnej_Precyzji is
8     new Ogolne_Funkcje_Zespolone
9              (Liczby_Zespolone_Podwojnej_Precyzji,
10              Funkcje_Elementarne_Podwojnej_Precyzji);
```

Drugą okolicznością, w której właściwe jest stosowanie formalnych parametrów ogólnych jest przypadek, w którym ta sama abstrakcja jest implementowana na kilka sposobów. Np. abstrakcja związku pomiędzy pewnym kluczem i wartością mu odpowiadającą (ang. *Mapping*) jest bardzo ogólna i łatwo wyobrazić sobie bardzo wiele sposobów jej realizacji. W większości przypadków abstrakcja ta jest scharakteryzowana przez typ klucza, typ wartości i operacje pozwalające związać te wielkości, usunąć to powiązanie oraz wyszukać wartość na podstawie klucza. Wszystkie kombinacje typów i operacji realizujące taki związek są dopuszczalne (każda taka kombinacja nazywana jest *sygnaturą*).

Pakiet ogólny może być używany do zdefiniowania, a następnie do konkretyzacji sygnatury. Skoro sygnatura jest zdefiniowana, formalny pakiet ogólny reprezentujący tę sygnaturę może być użyty jako „skrót” dla typu i operacji:

```
1  generic
2    -- definicja sygnatury
3    type Typ_Mapowania is limited private;
4    type Klucz is limited private;
```

```
5    type Wartosc is limited private;
6
7    with procedure Dodaj_Pare(M: in out Typ_Mapowania;
8                              K: in Klucz;
9                              V: in Wartosc);
10   with procedure Usuń_Pare (M: in out Typ_Mapowania;
11                             K: in Klucz;
12                             V: in Wartosc);
13   with procedure Zastosuj (M: in out Typ_Mapowania;
14                            K: in Klucz;
15                            V: in Wartosc);
16 package Mapowanie is
17 ...
18 end Mapowanie;
```

Następnie można zdefiniować taką jednostkę ogólną, która traktuje konkretyzację pakietu Mapowanie jako parametr, np.:

```
1  generic
2    with package Pewne_Mapowanie is new Mapowanie(<>);
3    with procedure Zrob_Cos_Z_Wartoscia(V: Pewne_Mapowanie.Wartosc)
4  procedure Zrob_Cos_Z_Kluczem(K: Pewne_Mapowanie.Klucz);
5
6  procedure Zrob_Cos_Z_Kluczem(K: Pewne_Mapowanie.Klucz) is
7    V: Pewne_Mapowanie.Wartosc;
8  begin
9    -- Znajdź wartość odpowiadającą kluczowi i zrób z nią coś
10   Pewne_Mapowanie.Zastosuj(K, V);
11   Zrob_Cos_Z_Wartoscia (V);
12 end Zrob_Cos_Z_Kluczem;
```

Czytelnik zauważył zapewne kłopotliwe powtarzanie Pewne_Mapowanie w treści jednostki ogólnej. Można tego uniknąć stosując klauzulę use:

```
generic
  with package Pewne_Mapowanie is new Mapowanie(<>);
  use Pewne_Mapowanie;
  with procedure Zrob_Cos_Z_Wartoscia (V: Wartosc)
procedure Zrob_Cos_Z_Kluczem(K: Klucz);
```

z odpowiednimi zmianami w treści pakietu.

Innym, bardziej matematycznym, przykładem jest sygnatura grupy.

```
1  generic
2    type Element_Grupy is private;
3    Identycznosc: in Element_Grupy;
4    with function Operacja(X, Y: Element_Grupy) return Element_Grupy
5    with function Odwrotność(X: Element_Grupy) return Element_Grupy;
6  package Sygnatura_Grupy is end;
```


Następujące funkcje ogólne dotyczą wielokrotnych operacji na elemencie grupy. Jeżeli prawy operand jest ujemny – zwracany jest wynik odwrotny, jeśli jest to zero to jest to operacja identyczności.

```
1  generic
```

```
2    with package Grupa is new Sygnatura_Grupy(<>);
3    use Grupa;
4  function Potega(X: Element_Grupy; N: Integer) return Element_Grupy;
5
6  function Potega(X: Element_Grupy; N: Integer) return Element_Grupy is
7    Wynik: Element_Grupy:= Identycznosc;
8  begin
9    for I in 1 .. abs(N) loop
10     Wynik := Operacja(Wynik, X);
11   end loop;
12   if N < 0 then
13     return Odwrotnosc (Wynik);
14   else
15     return Wynik;
16   end if;
17 end Potega;
```

Następująca konkretyzacja pozwala na deklarację grupy z operacją dodawania na ciele „długich" liczb zespolonych.

```
1  package Grupa_Dodawania_Liczb_Zespolonych_Podwojnej_Precyzji is
2    new Sygnatura_Grupy(
3            Element_Grupy => Liczby_Zespolone_Podwojnej_Precyzji.Zespolone,
4            Identycznosc => (0.0, 0.0);
5            Operacja => Liczby_Zespolone. "+";
6            Odwrotność => Liczby_Zespolone. "-");
```

Pozwala to na konkretyzację funkcji potęgowania dla tej grupy:

```
function Mnozenie_Liczb_Zespolonych is
  new Potega(Grupa_Dodawania_Liczb_Zespolonych_Podwojnej_Precyzji);
```

Oczywiście założono, że typ Zespolone nie jest typem prywatnym (rozdział 9.2) tak, że można zastosować agregat (rozdział 5) dla określenia elementu neutralnego.

13.2.1 Instrukcja wiążąca (synonim)

Niedoceniana i zbyt rzadko używana instrukcja wiążąca renames jest uogólnionym odpowiednikiem instrukcji with w Pascalu a także zmiennych referencyjnych w C++.

Jeżeli program posługuje się skomplikowanymi strukturami danych np.:

```
Y(fun1(2*a).pole1.pole2(fun2(b+a))).pole3;
```

i robi to wielokrotnie, to nie tylko zwiększa się prawdopodobieństwo pomyłki, ale i zmniejsza się efektywność programu. Dlatego też w języku Ada przewidziano system nadawania nazw dowolnym, istniejącym już obiektom bądź ich częściom np.:

```
r : Integer renames Y(fun1(2*a).pole1.pole2(fun2(b+a))).pole3;
```

Deklaracja taka **nie tworzy nowej zmiennej** a jedynie nazywa pewien element innej zmiennej. Zamiast odwoływać się do skomplikowanego wyrażenia,

można odwołać się po prostu do zmiennej r. W przeciwieństwie do innych języków programowania w Adzie można tworzyć synonimy dowolnych obiektów w programie np.:

```
procedure syn( a, b : integer) renames pakiet1.pakiet2.pakiet3.proc;
ex : exception renames pakiet1.pakiet2.pakiet3.ex;
```

itd.

13.3 Jeszcze o typach złożonych

W Adzie, koncepcja typów złożonych jest rozszerzona na zadania (rozdział 14) i monitory (rozdział 14.5). Typy złożone to takie typy, które mają parametry uściślające ich strukturę wewnętrzną.

Zezwolenie na deklarowanie (bo już nie używanie) wartości o nieokreślonych granicach jest szczególnie istotne w typach klasowych (rozdział 12) i prywatnych (formalnych) typach z dyskryminantem (<>) (rozdział 11.1), ponieważ takie typy mają nieokreślony zbiór dyskryminantów i dlatego są nieokreślone.

Typ prywatny może być oznaczony jako mający nieokreśloną liczbę dysktyminantów w następujący sposób:

```
type T(<>) is private;
```

13.3.1 Parametryzacja typów dyskryminantami wskaźnikowymi

Rozdział ten jest uzupełnieniem informacji zawartych w rozdziale 12.9.2.

Parametrami typów (dyskryminantami) są zwykle wartości typów dyskretnych lub zakresów. Mogą to być także typy wskaźnikowe. Zastosowanie takich dyskryminantów charakteryzuje się pewnymi ciekawymi własnościami. Istnieją dwie różne sytuacje. Dyskryminant (czyli parametr typu) może być nazwanym typem wskaźnikowym lub może być anonimowym typem wskaźnikowym. Dlatego można zadeklarować:

```
type R1(D: access T) is ...
type AT is access T;
type R2(D: AT) is ...
```

W takim wypadku dyskryminant R1 jest *dyskryminantem wskaźnikowym*, a dyskryminant R2 jest *nazwanym dyskryminantem wskaźnikowym*. Podobna nomenklatura dotyczy parametrów podprogramów, które mogą być *anonimowym typem wskaźnikowym* (np. A : access T lub *nazwanym typem wskaźnikowym* (np. A : Ptr).

Dyskryminanty typów wskaźnikowych mają kilka interesujących własności. Ze względu na to, że typ wskaźnikowy ma minimalne ograniczenia w stosunku do dyskryminanta, może on być zainicjowany w taki sposób, żeby wskazywał na otaczający obiekt lub inny obiekt o przynajmniej takim samym „czasie życia”

(np. będący w tym samym zasięgu widoczności) jak obiekt zawierający dyskryminant. Dyskryminanty typów wskaźnikowych mogą być stosowane tylko do typów ograniczonych. Trzeba tu podkreślić, że zadanie i monitor (obiekt chroniony) także mogą mieć dyskryminanty wskaźnikowe.

W sytuacji, w której obiekt może być na wielokrotnie połączonej liście, wtedy możliwa jest taka sytuacja, że jeden ze wskaźników pokazuje na następny. Tym niemniej często jest ważne, by także umożliwić dostęp do obiektu otaczającego to połączenie. Korzystając z dyskryminantów wskaźnikowych, związek pomiędzy połączonym w listę wskaźnikiem, a otaczającym obiektem może być zainicjowany jako domyślna inicjalizacja odpowiedniego połączenia.

Typ dziedziczący po danym może określać nowy zbiór dyskryminantów. Dla typów nierozszerzalnych te nowe dyskryminanty nie są traktowane jako rozszerzenie oryginalnego typu, ale raczej jako zmiana nazwy lub ograniczeń oryginalnego dyskryminantu. Skoro tak, takie dyskryminanty muszą być używane do określenia wartości jednego z oryginalnych dyskryminantów typu bazowego. Taki nowy dyskryminant jest mocno związany z dyskryminantem typu bazowego, który określa, ponieważ przy konwersji z typu bazowego nowe dyskryminanty biorą swoje wartości od takiego dyskryminanta, oczywiście o ile nie został przy tym zgłoszony wyjątek Constraint_Error. Model implementacyjny jest taki, że nowe dyskryminanty zajmują miejsce starych. Następujące deklaracje są dozwolone:

```
type S1(I: Integer) is ...;
type S2(I: Integer; J: Integer) is ...;

type T1(N: Integer) is new S1(N);
type T2(N: Integer) is new S2(N, 37);
type T3(N: Integer) is new S2(N, N);
```

Interesujący jest ten ostatni przypadek, ponieważ nowe dyskryminanty są mapowane w miejsce obu starych. Konwersja z typu S2 do typu T3 sprawdza, czy wartości obu dyskryminantów S2 są takie same. Praktyczne użycie nowych dyskryminantów w typach nierozszerzalnych jest takie, że pozwala to sparametryzować pełen typ, o ile odpowiadający mu typ prywatny jest typem z dyskryminantami.

```
   type T(D: DT) is private;
private
   type T(D: DT) is new S(D);
```

W przypadku typów rozszerzalnych (obiektów) można albo dziedziczyć wszystkie dyskryminanty lub dostarczyć całkowicie nowy ich zestaw. W tym drugim przypadku typ bazowy musi być typem ograniczonym a nowe dyskryminanty mogą (ale nie muszą) być używane do określenia ograniczeń. Dlatego typ rozszerzony może mieć więcej dyskryminantów niż typ bazowy, co nie jest prawdą w przypadku typów nierozszerzalnych. Przykład zastosowanie takiej konstrukcji znaleźć można w rozdziale 12.8.

13.4 Jeszcze jeden przykład

Jako ostatni przykład podamy pakiet służący do formatowania liczb podobny do funkcji znanej z języka C – printf. Czytelnik z całą pewnością zauważy, że pakiet jest uproszczony i zawiera wyłącznie opis formatowania dotyczący liczb całkowitych.

Część specyfikacyjna pakietu wygląda nastepująco:

```
generic
   type val is range <>;
package Print is

   Format_Error : exception;


   function Clear (s : String) return String;
   -- funkcja wycina podwójne %%
   -- %% -> na jeden znak %

   function "&" (
         S : String;
         A : val)
     return String;

   -- funkcja zastępuje napisy
   -- %% -> na jeden znak %
   -- %d -> integer na takiej liczbie pól jak trzeba
   -- %4d -> integer na 4 polach
   -- %#16#3d -> integer na 3 polach w notacji szesnastkowej
   -- %04d -> integer na 4 polach ale zamiast spacji są zera

end Print;
```

Część implementacyjna

```
with Text_IO; use Text_IO;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;

package body Print is


   package IIO is new Integer_IO (val);

   function Clear (s : String) return String is
      Wyn : String := s;
      Idx : Natural := 1;
      Count : Natural := s'Length;
   begin
      -- Funcja wycina znaki %%
      loop
         Idx := Index (Wyn (Idx .. Count), "%");
         if Idx = 0 then
            exit;
         elsif Idx = Count then
```

```
20              raise Format_Error;
21           end if;
22           Wyn (Idx + 1 .. Count - 1) := Wyn (Idx + 2 .. Count);
23           Idx := Idx + 1;
24           Count := Count - 1;
25        end loop;
26        return Wyn (1 .. Count);
27     end Clear;
28
29
30
31     function "&" (S : String; A : val) return String is
32        Idx_Start : Natural := 1;
33        Idx_End : Natural;
34        Leading_Zeros : Boolean := False;
35        Ins_Str : String (1 .. 20);
36        Podstawa : Boolean := False;
37        Pierwszy_Znak : Boolean := True;
38        Szerokosc_Pola : Integer := 0; -- niewyspecyfikowana
39        Baza : Integer := 0;
40
41     begin
42        loop
43           Idx_Start := Index (S (Idx_Start .. S'Last), "%");
44           if Idx_Start = 0 then
45              return S;
46           end if;
47           if Idx_Start = S'Last then
48              raise Format_Error;
49           end if;
50           if S (Idx_Start + 1) /= '%' then
51              exit;
52           end if;
53        end loop;
54        Idx_Start := Idx_Start - 1;
55        -- w tym miejscu wiadomo, że napis od 1 do idx_start będzie
56        -- kopiowany bez zmian
57        Idx_End := Idx_Start + 2;
58        loop
59           case S (Idx_End) is
60              when 'd' =>
61                 -- koniec
62                 exit;
63              when '0' .. '9' =>
64                 if not Podstawa then
65                    if Pierwszy_Znak then
66                       if S (Idx_End) = '0' then
67                          Leading_Zeros := True;
68                       end if;
69                    else
70                       Szerokosc_Pola
71                          := Szerokosc_Pola * 10
72                                + Character'Pos (S (Idx_End))
73                                - Character'Pos ('0');
```

```
74                      end if;
75                      Pierwszy_Znak := False;
76                  else
77                      Baza :=
78                          Baza * 10
79                              + Character'Pos (S (Idx_End))
80                              - Character'Pos ('0');
81                  end if;
82              when '#' =>
83                  Podstawa := not Podstawa;
84              when others =>
85                  raise Format_Error;
86          end case;
87          Idx_End := Idx_End + 1;
88      end loop;
89      if Baza = 0 then -- baza jest nieokreślona
90          Baza := 10;
91      end if;
92      if Szerokosc_Pola /= 0 then
93          IIO.Put (Ins_Str (1 .. Szerokosc_Pola), A, Baza);
94      else
95          IIO.Put (Ins_Str, A, Baza);
96          for idx in Ins_Str'Range loop
97              if Ins_Str (idx) /= '␣' then
98                  Szerokosc_Pola := Ins_Str'Last + 1 - idx;
99                  Ins_Str (1 .. Szerokosc_Pola) := Ins_Str (idx .. Ins_Str'Last);
100                 exit;
101             end if;
102         end loop;
103     end if;
104     if Leading_Zeros then
105         for Idx in 1 .. Szerokosc_Pola loop
106             if Ins_Str (Idx) = '␣' then
107                 Ins_Str (Idx) := '0';
108             end if;
109         end loop;
110     end if;
111     if Idx_Start = 0 then
112         if Idx_End = S'Last then
113             return Ins_Str (1 .. Szerokosc_Pola);
114         else
115             return Ins_Str (1 .. Szerokosc_Pola) & S (Idx_End + 1 .. S'Last);
116         end if;
117     else
118         if Idx_End = S'Length then
119             return Clear (S (1 .. Idx_Start)) & Ins_Str (1 .. Szerokosc_Pola);
120         else
121             return Clear (S (1 .. Idx_Start))
122                           & Ins_Str (1 .. Szerokosc_Pola)
123                           & S (Idx_End + 1 .. S'Last);
124         end if;
125     end if;
126 end "&";
127
```

```
128 end Print;
```

Natomiast użycie tego programu mogłoby wyglądać nastepująco:

```
1  with Text_IO; use Text_IO;
2  with Print;
3
4  procedure UsePrint is
5
6      package IPrint is new Print (Integer); use IPrint;
7
8      i : Integer := 3;
9  begin
10     Put_Line ( "Liczbe mozna zapisac tak: %d" & i);
11     Put_Line ( "albo tak: %3d, czy nawet tak: %03d albo tak: %#16#3d"
12                & i & i & i);
13 end UsePrint;
```

Wynik działania tego programu jest taki:

```
Liczbe mozna zapisac tak: 3
albo tak:  3, czy nawet tak: 003 albo tak: 16#3#
```

Oczywiście analogiczny pakiet można napisać dla liczb typów modularnych, zmiennoprzecinkowych i stałoprzecinkowych. Pakiet pokzany tutaj jest naszym zdaniem bardzo użyteczny, stosujemy go w programach pisanych przez nas samych i zalecamy to czytelnikowi. Zwracamy uwagę, że w przeciwieństwie do znanej z języka C funkcji printf **niemożliwe jest** podstawienie wartości niewłaściwego typu.

13.5 Ćwiczenia

◁ Ćwiczenie 13.1 ▷▷▷ Zadeklarować pakiet ogólny definiujący operacje macierzowe. Jedna z wersji tego pakietu powinna być parametryzowana typem zmiennoprzecinkowym, inna stałoprzecinkowym.

◁ Ćwiczenie 13.2 ▷▷▷ Porównać treści tych pakietów.

◁ Ćwiczenie 13.3 ▷▷▷ Dokonać konkretyzacji tego pakietu dla różnych typów.

Rozdział 14

Zadania

Bardzo często dogodnie jest podzielić program na współbieżnie (czyli wykonywane z podziałem czasu) lub równoległe (wykonywane na różnych procesorach lub komputerach) *zadania* czyli fragmenty kodu, które wykonywane są asynchronicznie (tj. niezależnie od siebie), ale od czasu do czasu komunikują się wymieniając pomiędzy sobą dane. Program zaprojektowany jako program wielozadaniowy jest łatwiejszy w konserwacji od programu jednozadaniowego, ponadto taki styl programowania umożliwia tworzenie *systemów ze stopniową degradacją*, czyli zaprojektowanych tak, że jeżeli w pewnym zadaniu pojawi się błąd to system jako taki nie przestanie działać, a co najwyżej straci nieco na swojej funkcjonalności. Co więcej zadanie, w którym pojawił się błąd może zostać wznowione, przez co system odzyska swoją funkcjonalność. Na odpowiednim sprzęcie zadania mogą być wykonywane równolegle, przez co program będzie wykonywany szybciej.

Główną jednak zaletą takiego podejścia do programu jest wydzielenie stosunkowo prostych zadań systemu, które mogą być pisane niezależnie od siebie. Dla każdego zadania istnieje jasno określony sposób komunikacji tego zadania z innymi zadaniami w systemie.

Język Ada, jako jeden z nielicznych zawiera w sobie mechanizm definiowania zadań i ich wzajemnej komunikacji[1]. Zadanie (ang. *task*) jest obiektem typu zadaniowego. W danym programie może istnieć więcej niż jeden obiekt danego typu zadaniowego. Wszystkie obiekty danego typu zadaniowego mają takie same *wejścia* (czyli definicję współpracy z innymi zadaniami) i mają ten sam kod, a w związku z tym wykonują ten sam algorytm (oczywiście na potencjalnie różnych danych). Różne obiekty zadaniowe tego samego typu mogą być parametryzowane przez użycie dyskryminantów. Oczywiście typy zadaniowe są **zawsze** typami ograniczonymi, czyli nie istnieją dla nich operacje porównania i podstawienia.

Zadanie jest tworzone tak samo jak każdy inny obiekt, może być zadeklarowane w części deklaracyjnej innego obiektu (procedury, pakietu, zadania, instruk-

[1] Również takie języki jak Modula-2, czy Java zawierają podobne mechanizmy, choć komunikacja odbywa się tam wyłącznie za pomocą monitorów opisanych tu w rozdziale 14.5

cji złożonej) lub może być utworzone dynamicznie poprzez alokator (instrukcję new). Jak wynika z powyższego zdania, zadania mogą być zagnieżdżane, tak samo jak obiekty każdego innego typu.

Wszystkie zadania utworzone przez deklarację odpowiednich obiektów lub tworzone dynamicznie są aktywowane równolegle. W żadnym wypadku nie można nic założyć w kwestii kolejności wykonywania zadań równocześnie (logicznie, bo zwykle zadania wykonywane są jednak synchronicznie z podziałem czasu ze względu na typowe, jednoprocesorowe środowisko wykonawcze programu). Zadanie, które utworzyło zadania potomne również wykonuje się równolegle z zadaniami potomnymi. Zadanie to jednak **nie może** się zakończyć (tj. zwolnić zajmowanych zasobów) do czasu zakończenia wszystkich zadań potomnych. Program główny, zadeklarowany jest jako procedura, należy go jednak traktować jako procedurę wywołaną przez zadanie z biblioteki systemowej. Zadanie się kończy jeśli:

- ⇝ zadanie wykonało swoją ostatnia instrukcję
- ⇝ zadanie jest gotowe do zakończenia
- ⇝ wszystkie zadania potomne są zakończone lub gotowe do zakończenia
- ⇝ zadanie zostało awaryjnie zakończone instrukcją abort.

Poniżej przedstawiono przykład demonstrujący tę zależność, na przykładzie zadań zdefiniowanych w rozdziale 14.1

```
1  declare
2    type Globalny is access Serwer;
3    A, B : Serwer;
4    G : Globalny;
5  begin
6    -- aktywacja zadania A i B
7    declare
8      type Lokalny is access Serwer;
9      X : Globalny := new Serwer; -- aktywacja zadania X.all
10     L : Lokalny := new Serwer; -- aktywacja zadania L.all
11     C : Serwer;
12   begin
13     -- aktywacja zadania C
14     G := X; -- G i X opisują to samo zadanie
15     ...
16   end; -- oczekiwanie na zakończenie C i L.all (ale nie X.all)
17   ...
18 end; -- oczekiwanie na zakończenie A, B, i G.all
```

14.1 Deklaracja zadania

Jak już wspomnieliśmy, może istnieć jeden obiekt pewnego typu zadaniowego i może też istnieć wiele obiektów tego typu. W tym pierwszym przypadku deklaruje się zadanie, a w drugim należy zadeklarować typ zadaniowy i w odpowiednim momencie należy utworzyć odpowiednie obiekty (statyczne lub dynamiczne).

Zadanie deklaruje się analogicznie jak podprogram, przy czym zamiast słowa kluczowego procedure lub function używa się słowa kluczowego task. Jeżeli z tym zadaniem mają się komunikować inne zadania (tak jest w znakomitej większości wypadków[2]) następuje deklaracja wejść, czyli procedur wykonywanych równocześnie przez oba zadania (klienta i serwera). Wejścia te mogą mieć parametry, których dotyczą dokładnie takie same reguły składniowe i logiczne jak procedur. Jeżeli kilka wejść jest do siebie bardzo podobnych, to można je zadeklarować jako rodzinę wejść, czyli strukturę przypominającą tablicę (ale nie będącą nią!). Poniżej przedstawiono różne deklaracje zadań:

```
1  task Uzytkownik; -- zadanie, które nie ma wejść
2
3  task Przegladacz is
4     entry Nastepny_Lexem(L : in Element_Leksykalny);
5     entry Nastepna_Akcja(A : out Akcja_Przegladacza);
6  end Przegladacz;
7
8  task Sterownik is
9     entry Zadanie(Poziom)(D : Element); -- rodzina wejść
10 end Sterownik;
```

W przypadku, gdy w programie ma istnieć wiele obiektów pewnego zadania, lub w przypadku, gdy zadania muszą być tworzone dynamicznie, należy utworzyć typ zadaniowy i w odpowiednim momencie programu utworzyć obiekt tego typu. Typ zadaniowy tworzy się poprzez wstawienie słowa kluczowego type pomiędzy słowo task a nazwę typu. Poza tym składnia pozostaje niezmieniona. Typy zadaniowe mogą mieć parametry (dyskryminanty) pozwalające na właściwe zainicjowanie takiego obiektu. Dyskryminantami rządzą te same reguły, które dotyczą obiektów innego typu (np. rekordów) zob. tez rozdział 14.1 Poniżej znajdują się przykłady deklaracji typów zadaniowych.

```
1  task type Serwer is
2     entry Nastepna_Usluga (WI : in Usluga);
3     entry Zakoncz_Dzialanie;
4  end Serwer;
5
6  task type Sterownik_Klawiatury(ID : Uklad := Polska) is
7     entry Czytaj(C : out Character);
8     entry Pisz (C : in Character);
9  end Sterownik_Klawiatury;
```

Zadanie zaczyna swoje działanie w momencie, w którym jego deklaracja znajdzie się w zasięgu widoczności dynamicznej struktury programu. W przypadku zadań zadeklarowanych jako typy zadaniowe w momencie utworzenia odpowiedniej zmiennej, w przypadku „zwykłych" zadań w momencie, kiedy deklaracja treści znajdzie się w zasięgu widoczności. Poniżej przedstawiono przykłady tworzenia obiektów zadaniowych na podstawie typów zadaniowych:

```
Agent : Serwer;
Terminal : Sterownik_Klawiatury(Arabska);
Pula : array(1 .. 10) of Sterownik_Klawiatury; -- uruchomiono 10 zadań
```

[2]Zadanie udostępniające pewne usługi nazywane jest serwerem

W przypadku dynamicznego tworzenia zadań odpowiednie deklaracje mogłyby być następujące:

```
type Klawiatura is access Sterownik_Klawiatury;
Terminal : Klawiatura := new Sterownik_Klawiatury(Jezyk);
```

14.2 Aktywacja zadania

Wykonanie pewnego zadania polega na wykonaniu odpowiadającej temu zadaniu treści. Początkowy okres działania zadania nazywany jest *aktywacją zadania* i składa się z wykonania części deklaracyjnej treści zadania. Jeżeli w tej części zadania zostanie zgłoszony i nieobsłużony wyjątek, to zadanie zostanie uznane za zakończone.

Wszystkie zadania pojawiające się w części deklaracyjnej pewnego obiektu zostaną aktywowane jednocześnie. Dla zadań tworzonych dynamicznie aktywacja jest dokonywana po przydzieleniu pamięci, ale przed powrotem z funkcji new[3]. Zadanie, które tworzy inne zadania i aktywuje je, zostaje zawieszone do momentu zakończenia aktywacji zadań potomnych. Jeżeli jedna z tych aktywacji się nie powiedzie to zostanie zgłoszony w tym zadaniu wyjątek Tasking_Error, chyba, że zadanie potomne zostanie usunięte awaryjnie instrukcją abort, przed zakończeniem swojej aktywacji. Poniżej przedstawiono przykład aktywacji:

```
1 procedure P is
2   A, B : Serwer; -- aktywacja zadania A i B
3   C : Serwer; -- aktywacja zadania C jednoczesna z aktywacją zadania A i B
4 begin
5   -- zadania A, B, C są aktywowane jednocześnie przed
6   -- wykonaniem piewszej instrukcji procedury P
7   ...
8 end;
```

14.3 Treść zadania

Treść zadania nie różni się specjalnie od treści podprogramu. Podstawowa różnica polega na możliwości stosowania w treści zadania (i tylko w treści zadania) instrukcji accept. Zwykle zadanie ma postać pętli wykonującej pewne instrukcje w nieskończoność lub – rzadziej – do momentu zaistnienia sytuacji, w której zadanie jest zbędne.

```
1 task body Serwer is
2   -- deklaracja zmiennych lokalnych zadania
3 begin
4   ...
5   loop
6     ...
7   end loop;
```

[3]Oznacza to tylko to, że zadanie zostanie uznane za gotowe do wykonania, nie oznacza jednak, że wykonanie zostanie choćby jedna instrukcja tego zadania. Z drugiej strony nie jest wykluczone, że zostanie wykonanych szereg jego instrukcji.

```
8     ...
9  end Serwer;
```

14.4 Komunikacja i synchronizacja

Jest dość oczywiste, że musi istnieć mechanizm pozwalający zadaniom komunikować się pomiędzy sobą i synchronizować swe działanie. Zwykle (ale nie zawsze) komunikacja jest powiązana z synchronizacją. Komunikacja i synchronizacja ma miejsce w następujących sytuacjach:

- ⇝ Zadania synchronizują się w czasie aktywacji i zakończenia (poprzednie punkty tego rozdziału).
- ⇝ Monitory (obiekty chronione) pozwalają na operacje synchronizowanego dostępu do współdzielonych danych.
- ⇝ Spotkania (ang. *rendezvous*) są używane do zapewnienia synchronicznej komunikacji pomiędzy parami zadań.
- ⇝ Bezpośredni dostęp do zmiennych współdzielonych jest dozwolony, ale wymaga specjalnych zabiegów zapewniających spójność danych.

Na marginesie wspominamy tu o innych metodach komunikacji zadań nie wymagających synchronizacji, takich jak asynchroniczne przesylanie komunikatów polegające na tym, że proces klienta wysyła wiadomość do procesu serwera i nie czekając ani chwili kontynuje swoje działanie. Wiadomość ta trafia do *skrzynki pocztowej* procesu serwera i tam będzie kiedyś obsłużona. Taki sposób komunikacji jest niezwykle użyteczny w przypadku, kiedy czas związany z samym procesem komunikacji jest znaczny — na przykład przy komunikacji sieciowej. Ponieważ jednak język Ada nie zawiera konstrukcji językowych wspomagających taki sposób wymiany informacji, to nie będziemy go dalej omawiać.

14.5 Monitory

W przypadku, gdy dwa lub więcej asynchronicznie (współbieżnie lub równolegle) działających zadań potrzebuje dostępu do tego samego zasobu systemowego (lub po prostu pewnej zmiennej globalnej) to wymagany jest pewien mechanizm wzajemnego wykluczania. Jedną z najbardziej znanych metod służących temu celowi są monitory (zwane w Adzie obiektami chronionymi). Monitor to zbiór podprogramów mających tę własność, że jeżeli pewne zadanie wykonuje jakikolwiek z nich, to każde inne zadanie próbujące wykonać dowolny z tych podprogramów zostanie zawieszone (jego wykonanie zostanie wstrzymane) do czasu, aż zadanie wykonujące podprogram monitora opuści go.

Trzeba pamiętać, że dostęp do zmiennych współdzielonych przez kilka procesów w sposób niesynchronizowany jest błędem bardzo trudnym do wykrycia, a jego efektem może być błędne działanie programu raz na jakiś czas, na przykład raz na dobe, czy nawet raz w miesiącu, ale za to błąd taki ma zwykle charakter krytyczny. Nasze doświadczenia pokazują, że fakt ten jest często niedoceniany przez programistów, nawet tych, którzy uważają się za doświadczonych.

14.5.1 Deklaracja monitora

Deklaracja monitora składa się ze słowa kluczowego protected, po którym następuje nazwa monitora i słowo kluczowe is. Analogicznie deklaracja typu monitorowego różni się tylko słowem kluczowym type przed nazwą monitora i ewentualnie parametrami (dyskryminantami). Na zbiór podprogramów zapewniających dostęp do pewnego zasobu współdzielonego składają się procedury, funkcje i bariery. Te ostatnie deklarowane są tak samo jak procedury, a różnica polega na tym, że procedury są zawsze aktywne, natomiast bariery można selektywnie wyłączać (tzn. uniemożliwiać ich wywołanie). Proces wywołujący nieaktywna barierę **nie** blokuje monitora, ale zostaje zawieszony do chwili, w której bariera zostanie uaktywniona. Taki proces oczekiwania nie jest oczekiwaniem aktywnym (ang. *busy wait*), tzn. że takie zadanie w systemie istnieje ale procesor się nim nie zajmuje. Bariery obliczane są dopiero w momencie zakończenia wykonywania procedury lub bariery przez inny proces.

Prócz podprogramów „publicznych", tj. takich, które są widoczne dla klientów pewnego monitora w sekcji prywatnej (zob. rozdział 9.2) można zadeklarować podprogramy lokalne dostępne tylko dla publicznych podprogramów tego monitora. Trzeba też w tym miejscu zadeklarować wszystkie zmienne, które maja być tym monitorem chronione i, poza szczególnymi przykładami, konieczne jest ich zainicjowanie. W treści monitora **nie można** deklarować żadnych zmiennych. Poniżej przedstawiono przykładową deklarację monitorów wraz z odpowiadającą im treścią:

```
1  protected Tablica_Wspoldzielona is
2    -- typy Indeks, Element, i Tablica_Elementow są typami globalnymi
3    function Skladnik (N : in Indeks) return Element;
4    procedure Ustaw_Skladnik(N : in Indeks; E : in Element);
5  private
6    -- deklaracja właściwego obiektu chronionego
7    Tablica : Tablica_Elementow(Indeks) := (others => Pusty_Element);
8  end Tablica_Wspoldzielona;
9
10 protected body Tablica_Wspoldzielona is
11
12   function Skladnik(N : in Indeks) return Element is
13   begin
14     return Tablica(N);
15   end Skladnik;
16
17   procedure Ustaw_Skladnik (N : in Indeks; E : in Element) is
18   begin
19     Tablica(N) := E;
20   end Ustaw_Skladnik;
21 end Tablica_Wspoldzielona;
```

Przykładowa deklaracja typu monitorowego i odpowiadającej mu treści:

```
1  protected type Zasob is
2    entry Zagarnij;
3    procedure Oddaj;
4  private
5    Zajety : Boolean := False;
```

```
6  end Resource;
7
8  protected body Zasob is
9
10   entry Zagarnij when not Zajety is
11   begin
12     Zajety := True;
13     ...
14   end Zagarnij;
15
16   procedure Oddaj is
17   begin
18     Zajety := False;
19     ...
20   end Oddaj;
21
22 end Zasob;
```

Przykłady deklaracji obiektów na podstawie pewnego typu monitorowego:

```
Maszyna : Zasob;
Znaczniki : array(1 .. 100) of Zasob;
```

14.5.2 Podprogramy monitora i ich wywołanie

Podprogramy monitora zapewniają wyłączny dostęp do zmiennych znajdujących się w monitorze. Ponieważ jednak w Adzie funkcje nie mogą mieć efektów ubocznych (rozdział 6.8) możliwe jest równoległe wywoływanie funkcji monitora (funkcjom monitora nie wolno zmieniać żadnych zmiennych chronionych monitorem)[4]. Wywołanie procedury lub bariery blokuje dostęp do monitora, a oba rodzaje podprogramów mogą zmieniać zmienne globalne monitora. Dla procesu wywołującego podprogram monitora wywołanie to, od strony logicznej, niczym nie różni się od wywołania każdego innego podprogramu. Jedyną różnicą jest to, że zadanie wywołujące taki podprogram może zostać zawieszone. Jest to implementacja klasycznego problemu czytelników i pisarzy (Ben-Ari 1996).

Język nie definiuje, w jakiej kolejności ustawiane są zadania czekające na wejście do danego monitora. Pewne reguły określone są w Aneksie D definicji języka (Intermetrics Inc. 1995*b*, Annex D). W tym miejscu można wspomnieć o najprostszej metodzie jaką jest ustawianie się w kolejności zgłoszeń. Jednakże w przypadku, jeśli zadania mają przypisane priorytety (wartości ustalające poziom „ważności" danego zadania dla systemu), taki algorytm może spowodować błąd inwersji priorytetu (Ben-Ari 1996, Kopetz 1998).

Przykłady wywołania wcześniej zadeklarowanych procedur monitora mogą być następujące:

```
1 a := Tablica_Wspoldzielona.Skladnik(7);
2 Tablica_Wspoldzielona.Ustaw_Skladnik(N, E);
3 Maszyna.Oddaj;
4 Znaczniki(34).Zagarnij;
```

[4] Jednakże jest to tylko możliwość, która jest lub nie jest zaimplementowana w danym systemie

Uwaga! Podprogramy monitora powinny być tak krótkie jak to tylko możliwe ze względu na płynność działania całego systemu związaną z możliwie krótkim czasem wykonywania wzajemnie wykluczających się fragmentów programu.

14.5.3 Błędy wykonania związane z monitorem

Podprogramy monitora są takimi samymi jak inne podprogramami, wykonującymi się w taki sam sposób jak odpowiednie podprogramy spoza monitora. Wywołanie procedury spoza monitora nie powoduje oczywiście zwolnienia monitora i nie pozwoli uruchomić się innym, potencjalnie czekającym zadaniom. Jednakże nie każda instrukcja może być wykonana w takim podprogramie. Mianowicie nie może być wykonana żadna taka funkcja, która może spowodować zablokowanie się monitora (czyli uniemożliwienie dalszego działania pewnego zadania, co spowodowałoby w konsekwencji uniemożliwienie działania innych zadań korzystających z tego monitora)[5]. Instrukcjami, które potencjalnie mogą spowodować takie działanie są:

- instrukcja select
- instrukcja accept
- wywołanie wejścia
- instrukcja delay
- instrukcja abort
- tworzenie lub aktywacja zadania
- wywołanie publicznego podprogramu tego samego monitora, który jest aktualnie wykonywany
- wywołanie podprogramu innego monitora, który zawiesza aktualne zadanie. Jeżeli monitor jest wolny, to wywołanie jest dozwolone

Jeżeli zostanie wykryta jedna z powyższych sytuacji zostanie zgłoszony wyjątek PROGRAM_ERROR. Gdyby tak się nie stało mogłoby nastąpić zakleszczenie, potencjalnie długo nie zauważone.

Trzeba tu wspomnieć, że pewne biblioteki systemowe mogą powodować wywoływanie procedur monitora. Najbardziej typowymi przykładami jest czytanie z klawiatury, myszy, pisanie na ekran, operacje na plikach.

14.5.4 Cechy monitorów

Skalowalność Monitory pozwalają na efektywną implementację mechanizmu synchronizacji zarówno na komputerze jednoprocesorowym jak i wieloprocesorowym.

[5] Zjawisko takie nazywane jest *łańcuchowym zakleszczeniem* (ang. *chain deadlock*)

Adaptowalność Dodatkowe podprogramy monitora mogą być dodawane bez potrzeby modyfikacji istniejących aplikacji korzystających z tego monitora.

Modularność Wszystkie metody dostępu do zmiennych współdzielonych są ściśle określone i niemożliwe do zmiany w inny niż zamierzony sposób.

Wydajność Rozmiar i wymagania inicjacyjne związane z monitorami są doskonale znane w czasie kompilacji zadania korzystającego z tego monitora. Pozwala to na to, by dane związane z monitorem umieszczać na stosie i w prosty sposób je inicjować. Ponieważ podprogramy monitora nie mogą w praktyce korzystać z innych monitorów (zob. wyżej) bardzo upraszcza się algorytm ustalania „zajętości" monitora.

Czytelność Specyfikacja obiektu wyraźnie odróżnia podprogramy, które mogą zmieniać stan zmiennych monitora (procedury), które mogą być zablokowane (bariery), od tych, które mogą tylko czytać ten stan (funkcje). Odróżnienie to jest szczególnie istotne w systemach czasu rzeczywistego ze względu na łatwiejszą analizę poprawności takiego programu, w którym na pewno nie pojawi się zakleszczenie (pojawienie się sytuacji zagrażającej zakleszczeniem sygnalizowane jest zgłoszeniem wyjątku PROGRAM_ERROR).

Kompatybilność z koncepcją spotkań Bariera w monitorze odpowiada dozorowi, procedura i funkcja – odpowiedniej gałęzi instrukcji select.

Obsługa przerwań Monitory są znakomicie przystosowane do współpracy z systemem przerwań (bezparametrowa procedura monitora może być użyta jako bezpośrednia procedura obsługi przerwania). Zarówno procedura obsługi przerwania jak i podprogram monitora powinien być napisany tak, by odpowiedni kod wykonywany był możliwie jak najszybciej. Ponadto koncepcję monitorów łatwo związać z systemem priorytetów.

W najprostszym jednoprocesorowym środowisku, gdy nie jest zaimplementowany mechanizm priorytetów, monitor może być zaimplementowany poprzez proste zablokowanie szeregowania zadań w czasie, gdy jedno z nich wykonuje podprogram monitora. W przypadku, w którym zadania mają wiele priorytetów, powinien być zastosowany mechanizm podwyższania priorytetów (ang. *ceiling priorities*) (Intermetrics Inc. 1995*b*, Kopetz 1998),w którym tylko zadania o pewnym priorytecie i niższym mogą używać danego monitora i wtedy zadanie o priorytecie wyższym może bezpiecznie przerwać wykonanie podprogramu monitora, ponieważ nie będzie go używać.

14.5.5 Przykłady użycia

Implementacja synchronizacji

W poniższym przykładzie założono, że wiele zadań oczekuje na zaistnienie pewnego zdarzenia. Gdy zdarzenie takie zaistnieje, to pewne inne zadanie to zasygnalizuje. Sygnał powoduje, że **tylko jedno** zadanie zostanie uruchomione w odpowiedzi na zgłoszenie. Jeżeli żaden proces nie czeka na sygnał, to w przypadku, gdy pierwszy proces zgłosi chęć oczekiwania na ten sygnał, będzie on

obsługiwany natychmiast. Wiele sygnałów zgłoszonych w czasie, gdy nikt na nie nie czeka traktuje są jak jeden sygnał.

```
protected Zdarzenie is
   entry Czekaj; -- czekaj na zdarzenie
   procedure Sygnalizuj; -- sygnał, ze zdarzenie sie pojawiło.
private
   Pojawilo_Sie : Boolean := False;
end Zdarzenie;

protected body Zdarzenie is

   entry Czekaj when Pojawilo_Sie is
   begin
      Pojawilo_Sie:= False; -- konsument sygnału
   end Czekaj;

   procedure Sygnalizuj is
   begin
      Pojawilo_Sie := True; -- Ustawia barierę Wait na TRUE
   end Sygnalizuj;

end Zdarzenie;
```

Zadanie czeka na zdarzenie wywołując barierę

```
Zdarzenie.Czekaj;
```

natomiast inne zadanie sygnalizuje zdarzenie poprzez wywołanie

```
Zdarzenie.Sygnalizuj;
```

Jeżeli przed wywołaniem Sygnalizuj wywołane będzie Czekaj to okaże się, że bariera Czekaj jest zablokowana poprzez początkową wartość zmiennej Pojawilo_Sie. Wywołanie procedury Sygnalizuj zmieni wartość zmiennej zezwalając procedurze Czekaj zająć monitor i powtórnie zmienić wartość zmiennej Pojawilo_Sie.

Jeżeli przed wywołaniem procedury Sygnalizuj inne procesy będą oczekiwać na barierze Czekaj, to w przypadku pojawienia się sygnału, pierwszy proces oczekujący na barierze zostanie zwolniony i przestawi zmienną Pojawilo_Sie na FALSE, co zapobiegnie przejściu przez barierę przez następne zadanie.

Podkreślamy tu, że w Adzie, w przeciwieństwie do Moduli-2 czy Javy, sprawdzenie, czy monitor jest zajęty następuje *przed* wejściem do monitora, a nie dopiero w nim samym.

Ogólny bufor ograniczony

Przykład ten definiuje podstawowe operacje na buforze ograniczonym, w którym typ elementów bufora jest typem ogólnym. Operacje na buforze składają się z barier Send (umieszczającej element w buforze, o ile jest na niego miejsce) i Receive (pobierającej element z bufora, o ile jest w nim jakiś element).

```
generic
   type Element is private;
```

```
3     Wielkosc_Skrzynki: in Natural;
4  package Pakiet_Skrzynki_Pocztowej is
5
6     type Liczba_Elementow is range 0 .. Wielkosc_Skrzynki;
7     type Indeks_Elementu is range 1 .. Wielkosc_Skrzynki;
8     type Tablica_Elementow is array (Indeks_Elementu) of Element;
9
10    protected type Skrzynka_Pocztowa is
11       entry Wyslij(Elem: Element);
12       entry Odbierz(Elem: out Element);
13    private
14       Liczba_Elementow : Liczba_Elementow := 0;
15       Indeks_Wyjsciowy : Indeks_Elementu := 1;
16       Indeks_Wejsciowy : Indeks_Elementu := 1;
17       Dane : Tablica_Elementow;
18       -- inicjacja nie jest tu konieczna
19    end Skrzynka_Pocztowa;
20 end Pakiet_Skrzynki_Pocztowej;
```

Treść monitora może być następująca:

```
1  package body Pakiet_Skrzynki_Pocztowej is
2     protected body Skrzynka_Pocztowa is
3        entry Wyslij(Elem: Element)
4              when Liczba_Elementow < Wielkosc_Skrzynki is
5        -- blokuj jeśli nie ma miejsca w buforze
6        begin
7           Dane(Indeks_Wejsciowy) := Elem;
8           Indeks_Wejsciowy := Indeks_Wejsciowy mod Wielkosc_Skrzynki + 1;
9           Liczba_Elementow := Liczba_Elementow + 1;
10       end Wyslij;
11
12       entry Odbierz(Elem: out Element) when Liczba_Elementow > 0 is
13          -- blokuj jeśli bufor jest pusty
14       begin
15          Elem := Dane (Indeks_Wyjsciowy);
16          Indeks_Wyjsciowy:= Indeks_Wyjsciowy mod Wielkosc_Skrzynki + 1;
17          Liczba_Elementow := Liczba_Elementow - 1;
18       end Odbierz;
19    end Skrzynka_Pocztowa;
20 end Pakiet_Skrzynki_Pocztowej;
```

Ujemną stroną takiej deklaracji jest to, że typ Tablica_Elementow musi być zadeklarowany poza monitorem.

14.5.6 Czas

Język Ada zwiera dwie wersje instrukcji, których zadaniem jest zawieszenie działania aktualnego zadania na pewien okres czasu. Jedną z nich jest instrukcja delay z parametrem typu Duration (jest to typ stałoprzecinkowy zależny od implementacji), która powoduje zawieszenie danego zadania na co najmniej tyle sekund ile określa parametr tej instrukcji. Dokładniej rzecz ujmując po upływie określonego czasu zadanie staje się gotowe do wykonania i po czasie wynikającym z

obciążenia systemu zadanie to zostanie uruchomione. Dokładność odmierzania czasu zależy od systemu i jest określona dokładnie w (Intermetrics Inc. 1995*b*, §D.9).

Drugą z instrukcji zawieszających wykonanie aktualnego zadania jest instrukcja delay until z parametrem typu Time, określająca czas, w którym zadanie ma wznowić swoje działanie. W przypadku programu mającemu za zadanie wykonać procedurę Pewna_Operacja, różnica pomiędzy programem:

```
loop
  Pewna_Operacja;
  delay 1.0;
end loop;
```

a programem pokazanym poniżej, w którym odpowiednie typy danych opisane są w bibliotece systemowej Ada.Real_Time:

```
Okres : Time_Span := 1.0;
Czas_Wykonania : Time := Clock+Okres;
loop
  Pewna_Operacja;
  delay until Czas_Wykonania;
  Czas_Wykonania := Czas_Wykonania+Okres;
end loop;
```

jest następująca:

W pierwszej wersji programu, w dłuższym okresie czasu (np. w ciągu godziny) procedura Pewna_Operacja będzie wykonana kilka razy mniej niż w drugim przypadku ze względu na sumowanie się opóźnień.

14.6 Spotkania

Prócz komunikacji za pomocą zmiennych globalnych realizowanej za pomocą monitorów, w Adzie istnieje możliwość bezpośredniej komunikacji zadań za pomocą mechanizmu nazywanego *spotkaniem* (ang. *rendez-vous*). Mechanizm ten polega na tym, że zadanie udostępniające usługę i zadanie chcące z tej usługi skorzystać czekają w pewnym punkcie programu na siebie i następnie wykonywany jest wspólny fragment kodu dla obu tych zadań. Po wykonaniu tego fragmentu kodu zadania biorące udział w spotkaniu kontynuują swoje działanie asynchronicznie.

Zadanie udostępniające pewną usługę w sposób wyspecyfikowany w deklaracji zadania w deklaracji wejścia (entry) udostępnia ją wykonując instrukcję accept z nazwą i parametrami odpowiadającymi deklaracji. W ten sposób zdefiniowany jest wspólny fragment kodu dla obu zadań biorących udział w spotkaniu. Instrukcja accept ma następująca składnię:

```
1 accept Nazwa_Wejscia do
2   Ciag_Instrukcji;
3   Sekcja_Obslugi_Wyjatkow;
4 end Nazwa_Wejscia.
```

Wyjątek zgłoszony i nieobsłużony w treści instrukcji accept jest propagowany **do obu zadań** biorących udział w spotkaniu.

Poniżej przedstawiono deklaracje wejść i odpowiadające im instrukcje accept, wykonywane w zadaniu serwera.

```
entry Czytaj(V : out Element);
entry Zakoncz_Dzialanie;
entry Zadaj(Poziom)(D : Element); -- rodzina wejść

accept Zakoncz_Dzialanie; -- "pusta" instrukcja accept
accept Czytaj(V : out Element ) do
  V := Element_Lokalny;
end Czytaj;
accept Zadaj(Niski)(D : Element) do -- rodzina wejść
  ...
end Zadaj;
```

Wywołanie wejścia jest takie samo jak wywołanie procedury i składa się z nazwy zadania oddzielonej kropką od nazwy wejścia wraz z odpowiednimi parametrami. Różnica polega na tym, że w przypadku, gdy zadanie serwera jest jeszcze niegotowe do spotkania zadanie wywołujące wejście zostanie zawieszone, podobnie jak to ma miejsce w przypadku próby wywołania podprogramu zajętego monitora lub niekatywnej bariery.

Przykłady wywołania wejścia

```
Agent.Zakoncz_Dzialanie;
Przegladacz. Nastepny_Lexem (E);
Pula(5).Czytaj(Nastepny_Znak);
Maszyna.ZAdaj(Niski)(Pewien_Element);
Znaczniki(3).Zagarnij;
```

Trzeba tu zwrócić uwagę na to, że spotkanie jest operacja asymetryczną. Zadanie wywołujące wejście „wie" z kim się spotyka, natomiast zadanie przyjmujące wywołanie nie może tego ustalić. Jeżeli jednak byłoby konieczne ustalenie tego faktu, zadanie serwera może skorzystać z pakietu standardowego Ada.Task_Identification.

14.7 Instrukcja `select`

Ponieważ wywołanie wejścia, a także przyjęcie wejścia (a w zasadzie gotowość do wykonania takiej czynności), jest operacją bezwarunkową, to w momencie, w którym jedno z zadań wykona tę operacje na pewno nie będzie w stanie wykonać żadnej innej operacji aż do momentu dokończenia spotkania. W przypadku wystąpienia awarii powodowałoby to łańcuchowe zakleszczenie. Dlatego też działanie instrukcji accept może być modyfikowane instrukcją select. Instrukcja ta ma cztery formy, z których jedna pozwala oczekiwać na wiele różnych spotkań, dwie formy pozwalają na uzyskanie warunkowego wywołania wejścia, w tym przeterminowania operacji (zaniechania operacji, która się nie dokonała w określonym czasie). Ostatnia dostarcza mechanizmu zwanego asynchroniczną zmianą wątku sterowania.

14.7.1 Selektywne oczekiwanie

Omawiana forma instrukcji select pozwala na tworzenie konstrukcji w zadaniu serwerze, pozwalających na oczekiwanie na wiele spotkań jednocześnie.

Instrukcja ta ma następującą formę:

```
1 select
2   Instrukcja_accept;
3 or
4   Instrukcja_accept;
5 or
6   Instrukcja_accept;
7 ...
8 end select;
```

Jeżeli zadanie klienta wywoła **dowolne** wejście, to instrukcja select zostaje uznana za wykonaną i klienci wywołujący inne wejścia objęte tą instrukcją muszą czekać na ponowne jej wywołanie. Wszystkie wejścia są równoprawne. Oczywiście instrukcja select musi posiadać co najmniej jedną instrukcję accept.

W przypadku, w którym w momencie wykonania instrukcji select zadania klientów oczekują w więcej niż jednym wejściu, to wybór tego, które wejście zostanie przyjęte zależy od przyjętego mechanizmu szeregowania.

14.7.2 Dozory

Instrukcja accept może mieć rozszerzoną składnię używaną w ramach instrukcji select, a służącą do okresowego blokowania wejścia. Taką wersję instrukcji accept nazywa się *dozorem*. Instrukcja ta ma następującą składnię

```
when Warunek => accept Nazwa( parametry ) do
```

Wejście tak określone może być przyjęte tylko wtedy, gdy warunek ma wartość TRUE. Jeśli warunek skojarzony z pewnym wejściem ma wartość FALSE, to zadanie wywołujące dane wejście zostanie zawieszone do momentu ponownego wywołania instrukcji select, w której warunek będzie miał wartość TRUE. Warunki obliczane są wyłącznie w momencie wykonywania instrukcji select, zmiana wartości pewnych zmiennych, które mają wpływ na obliczanie warunku wewnątrz instrukcji select **nie powoduje** ponownego obliczenia żadnego z nich. Czytelniku lub Czytelniczko, czy dostrzegasz podobieństwo do barier w monitorach? Tak? To bardzo dobrze, gdyż są to mechanizmy bardzo podobne, chociaż zdecydowanie nie takie same.

W sytuacji, w której wszystkie gałęzie instrukcji select mają zablokowane dozory, następuje zakleszczenie, ponieważ nie ma możliwości wykonania tej instrukcji. Oczywiście nie można też wykonać instrukcji accept z dozorem bez instrukcji select z tego samego powodu. Uważny czytelnik zwróci uwagę na podobieństwo instrukcji select do monitora. Gałęzie bez dozorów odpowiadają w przybliżeniu procedurom monitora, gałęzie z dozorami odpowiadają barierom. Jednakże trzeba zdawać sobie sprawę z tego, że implementacja tego samego mechanizmu (zob. przykład zadania Skrzynka_Pocztowa poniżej) w postaci zadania jest mniej efektywna niż implementacja monitora, zajmuje też więcej

miejsca w pamięci choćby ze względu na stos procesu. Jednakże w przypadku, w którym wymagana jest bardziej skomplikowana inicjacja (procedura Pewne_Instrukcje_Inicjujace_Zadanie) lub też po skomunikowaniu się wymagane są inne operacje – po wywołaniu odpowiedniego wejścia wywoływana jest asynchronicznie procedura Ciag_Instrukcji_Wykonywanych_Po_Wywolaniu_Wejscia_Send albo Ciag_Instrukcji_Wykonywanych_Po_Wywolaniu_Wejscia_Receive, lub po wywołaniu dowolnego wejścia – procedura Inne_Instrukcje, zadanie jest rozwiązaniem prostszym i pewniejszym. Poniżej przedstawiono rozwiązanie funkcjonalnie podobne do przedstawionego na stronie 314 pakietu Skrzynka_Pocztowa.

```
1  task type Skrzynka_Pocztowa is
2    entry Wyslij(Elem: Element);
3    entry Odbierz(Elem: out Elemet);
4  end Skrzynka_Pocztowa;
```

Treść zadania odpowiadająca takiej deklaracji:

```
1  task body Skrzynka_Pocztowa is
2    Liczba_Elementow : Liczba_Elementow := 0;
3    Indeks_Wyjsciowy : Indeks_Elementu := 1;
4    Indeks_Wejsciowy : Indeks_Elementu := 1;
5    Dane : Tablica_Elementow;
6  begin
7    Pewne_Instrukcje_Inicjujace_Zadanie;
8    loop
9      select
10       when Liczba_Elementow < Wielkosc_Skrzynki =>
11         accept Wyslij(Elem: Element) do
12           Dane(Indeks_Wejsciowy) := Elem;
13           Indeks_Wejsciowy := Indeks_Wejsciowy mod Wielkosc_Skrzynki+1;
14           Liczba_Elementow := Liczba_Elementow + 1;
15         end Wyslij;
16         Ciag_instrukcji_Wykonywanych_Po_Wywolaniu_Wejscia_Send;
17       or
18         when Liczba_Elementow > 0 =>
19           accept Odbierz(Elem: out Item) do
20             Elem := Dane(Indeks_Wyjsciowy);
21             Indeks_Wyjsciowy := Indeks_Wyjsciowy mod Wielkosc_Skrzynki+1;
22             Liczba_Elementow := Liczba_Elementow−1;
23           end Odbierz;
24           Ciag instrukcji_Wykonywanych_Po_Wywolaniu_Wejscia_Receive;
25       end select;
26       Inne_Instrukcje;
27     end loop;
28 end Skrzynka_Pocztowa;
```

Warto zauważyć, że gdyby zmienne Indeks_Wejsciowy i Indeks_Wyjsciowy były typu modularnego (rozdział 11.5), a nie całkowitoliczbowego (rozdział 3.7), to zamiast instrukcji Indeks := Indeks mod Wlk + 1 moglibyśmy napisać Indeks := Indeks + 1. Przewijanie nastąpiłoby automatycznie.

14.7.3 Dodatkowe własności instrukcji `select`

Prócz wejść istnieje możliwość włączania w instrukcję select następujących gałęzi innego typu:

- ⇝ gałęzi terminate (tylko jednej)
- ⇝ jednej lub większej ilości gałęzi delay delay until
- ⇝ gałęzi alternatywnej (tj. słowa kluczowego else z następującym po nim ciągiem instrukcji.

Te trzy możliwości wykluczają się wzajemnie i oczywiście są tylko uzupełnieniem instrukcji accept, które muszą pojawić się w instrukcji select.

Gałąź terminate zgłasza gotowość zakończenia zadania (ale go nie kończy!). Gdyby w zadaniu Skrzynka_Pocztowa znalazła się gałąź terminate:

```
1  select
2    when Liczba_Elementow < Wielkosc_Skrzynki =>
3      accept Wyslij(Elem: Item) do
4        ...
5      end Wyslij;
6  or
7    when Liczba_Elementow > 0 =>
8      accept Odbierz(Elem: out Item) do
9        ...
10     end Odbierz;
11 or
12   terminate;
13 end select;
```

to, w przypadku, gdy żadne inne zadania nie wywołują wejść Wyslij i Odbierz, skończą się wszystkie ewentualne zadania potomne tego zadania, i skończy się zadanie nadrzędne, to zadanie Skrzynka_Pocztowa również zostanie zakończone. Konstrukcja taka jest użyteczna ze względu na to, żeby można było w prosty sposób, niejawnie zakończyć wszelkie zadania potomne, a trzeba pamiętać, że programista niekoniecznie musi wiedzieć, o tym, że program, ktory tworzy jest programem wielozadaniowym, ponieważ mogą istnieć zadania uruchamiane przez pakiety biblioteczne, które, gdyby nie zawierały gałęzi terminate powodowałyby, że procedura stanowiąca program główny nie mogłaby się zakończyć. Stąd wynika ważna uwaga dla programistów tworzących biblioteki: Jeżeli w Twojej bibliotece znajduje się zadanie, i nie ma innych przeciwwskazań to nie żałuj tych kilku znaków – dopisz gałąź terminate.

Gałąź delay jest używana wtedy, kiedy zadanie serwera ma oczekiwać na dowolne wywołanie tylko określony czas (lub do określonego czasu – delay until). Gdyby w zadaniu Skrzynka_Pocztowa znalazły się gałęzie delay:

```
1  select
2    when Liczba_Elementow < Wielkosc_Skrzynki =>
3      accept Wyslij(Elem: Item) do
4        ...
5      end Wyslij;
```

```
or
  when Liczba_Elementow > 0 =>
    accept Odbierz(Elem: out Item) do
      ...
    end Odbierz;
or
  delay 5.0;
  Instrukcje_Wykonywane_Po_Oczekiwaniu_5;
or
  delay Count*0.3;
  Instrukcje_Wykonywane_Po_Oczekiwaniu_Obliczanym;
end select;
```

to, w przypadku, gdy żadne inne zadania nie wywołują wejść Wyslij i Odbierz i upłynie najkrótszy czas wynikający ze wszystkich instrukcji delay (ewentualnie delay until) zadanie przejdzie do wykonania procedury Instrukcje_Wykonywane_Po_Oczekiwaniu_5 lub Instrukcje_Wykonywane_Po_Oczekiwaniu_Obliczanym, w zależności od tego, która z instrukcji delay zostanie wywołana. W przypadku, w którym kilka instrukcji delay zakończy swoje działanie jednocześnie, wybór tej procedury zależy od implementacji. Odliczanie czasu zaczyna się w momencie obliczenia dozorów. **Nie można używać jednocześnie oczekiwania względnego (**delay**) i bezwzględnego (**delay until**)**. Uważny czytelnik z pewnością zauważy, że nie zmniejsza to wcale ogólności konstrukcji.

Gałąź alternatywna (gałąź else) jest równoważna logicznie instrukcji delay 0.0. Gdyby w zadaniu Skrzynka_Pocztowa znalazła się gałąź alternatywna:

```
select
  when Liczba_Elementow < Wielkosc_Skrzynki =>
    accept Wyslij(Elem: Item) do
      ...
    end Wyslij;
or
  when Liczba_Elementow > 0 =>
    accept Odbierz(Elem: out Item) do
      ...
    end Odbierz;
else
  Instrukcje_Alternatywne;
end select;
```

to, w przypadku, gdyby żaden klient nie wywołał ani wejścia Wyślij ani Odbierz **przed** wykonaniem instrukcji select, to instrukcja ta, nie czekając ani chwili przeszłaby do wykonania procedury Instrukcje_Alternatywne.

14.7.4 Wywołanie wejścia z przeterminowaniem

Jak to pokazano w poprzednim punkcie zadanie serwera może przestać oczekiwać na spotkanie, w przypadku, jeśli nie dochodzi do niego zbyt długo. Taki sam mechanizm musi być zastosowany również po stronie klienta. Wszystkie szczegóły implementacyjne są takie same jak w przypadku przyjęcia wejścia z przeterminowaniem.

Poniższe przykłady pokazują fragmenty programów implementujących ten mechanizm.

```
select
   Maszyna.Zadanie(Sredni)(Pewien_Element);
or
   delay 45.0;
   -- sterownik zajęty, trzeba zrobic coś innego
end select;

select
   Skrzynka_Pocztowa.Odbierz( it );
   Ciag_Instrukcji_Po_Wywolaniu_Wejscia;
or
   delay until Pewien_Czas;
   Ciag_Instrukcji_Po_Przeterminowaniu;
end select;
```

14.7.5 Warunkowe wywołanie wejścia

Analogicznie do przykładu działań alternatywnych podejmowanych przez zadanie – serwer w przypadku, w którym klient jest niegotowy (rozdział 14.7.3), również zadanie – klient może natychmiastowo zrezygnować ze spotkania, w przypadku, gdy serwer jest niegotowy. Sposób realizacji tego zadania demonstruje następujący przykład realizacji aktywnego oczekiwania, czyli takiego sposobu programowania, którego **należy unikać**:

```
procedure Przegladaj(R : in Zasob) is
begin
   loop
      select
         R.Zagarnij;
         return;
      else
         null; -- aktywne oczekiwanie
      end select;
   end loop;
end Przegladaj;
```

Inny przykład prostego programu testującego warunkowe wywołanie wejścia:

```
with Text_IO; use Text_IO;

procedure t1 is

   task p is
      entry pe;
   end p;

   task body p is
   begin
      loop
         accept pe;
```

```
13              delay 10.0;
14          end loop;
15      end p;
16
17  begin
18      loop
19          select
20              p.pe;
21              Put ('*');
22          or
23              delay 5.0;
24              Put ('#');
25          end select;
26      end loop;
27  end t1;
```

a wynik działania tego programu:

###*#*#*#*#

Warto zwrócić uwagę na to, że w pewnym momencie pojawiają się dwa znaki # obok siebie, co jest wynikiem sposobu szeregowania zadań.

Asynchroniczna zmiana wątku sterowania

Asynchroniczna zmiana wątku sterowania (ang. *Asynchronous Transfer of Control*) pozwala na przekazanie sterowania z pewnego ciągu instrukcji do innego, jeżeli pierwotnie wykonywane instrukcje wykonują się zbyt długo lub nastąpi zewnętrzny sygnał (wywołanie wejścia) przerywającego ich wykonanie. Składnia tej instrukcji jest następująca:

```
select
   Instrukcja_Wyzwalająca
then abort
   Ciag_Instrukcji
end select;
```

Instrukcją wyzwalającą może być instrukcja delay:

```
delay 5.0;
Instrukcje_Wykonywane_Po_Delay;
-- Ta część programu zostanie wykonana wtedy i tylko wtedy,
-- gdy "Ciag_Instrukcji" nie zdąży się zakończyć w ciągu 5 sekund
```

lub

```
delay until Pewien_Czas;
Instrukcje_Wykonywane_Po_Delay_Until;
-- Ta część programu zostanie wykonana wtedy i tylko wtedy,
-- gdy "Ciag_Instrukcji" nie zdąży się zakończyć do momentu
-- w którym upłynie "Pewien_Czas"
```

lub

```
accept wej( ... );
Instrukcje_Wykonywane_Po_Sygnale_Wej;
-- Ta część programu zostanie wykonana wtedy i tylko wtedy,
-- gdy "Ciag_Instrukcji" nie zdąży się zakończyć
-- przed wywołaniem wejścia "Wej"
```

W tym ostatnim przypadku, jeżeli w momencie wykonania instrukcji select pewne zadanie już czeka z wywołaniem wejścia Wej to Ciag_Instrukcji zostanie przerwany zanim się zacznie, czyli nie będzie się wykonywał ani chwili.

Przykład interpretera rozkazów:

```
loop
   select
      Terminal.Czekaj_Na_Przerwanie;
      Put_Line("Przerwanie");
   then abort
    -- Ta część programu będzie przerwana sygnałem terminala
    Put_Line ("->␣");
    Get_Line (Komenda, Ostatnii);
    Przetwarzaj_Komende (Komenda (1..Ostatni));
   end select;
end loop;
```

Przykład obliczeń ograniczonych czasem:

```
select
   delay 5.0;
   Put_Line("Obliczenia␣nie␣sa␣zbiezne");
then abort
   -- Ta procedura powinna się liczyć nie dłużej niż 5.0 sekund;
   -- jeśli nie – algorytm jest rozbieżny
   Bardzo_Skomplikowana_Funkcja(X, Y);
end select;
```

14.8 Awaryjne usunięcie zadania – instrukcja `abort`

Instrukcja abort powoduje usunięcie zadania, które zachowuje się w sposób nienormalny i nie daje się zakończyć w zwykły sposób (tj. przez wykonanie ostatniej instrukcji zadania). Celem takiej operacji jest uniemożliwienie temu zadaniu dalszej, szkodliwej interakcji z systemem, ponieważ „niedziałanie” programu jest znacznie bardziej bezpieczną sytuacją niż złe działanie tego programu. Instrukcja abort jest bardzo prosta, jej przykładem mogą być:

```
abort Skrzynka_Pocztowa;
abort Maszyna, Rodzina_Zadan(7);
```

Zadanie określone parametrem instrukcji abort usuwane jest natychmiast przerywając wykonanie w dowolnym miejscu z wyjątkiem następujących:

- wykonanie procedury monitora;

- ⇝ oczekiwanie na zakończenie wejścia
- ⇝ oczekiwanie na zakończenie zadań zależnych
- ⇝ Wykonanie procedury Initialize jako ostatniego kroku w inicjacji nadzorowanego obiektu (rozdział 12.7)
- ⇝ Wykonanie procedury Finalize jako ostatniego kroku w usuwaniu nadzorowanego obiektu (rozdział 12.7)
- ⇝ Wykonywanie podstawienia nadzorowanego obiektu (rozdział 12.7)

Kiedy zadanie jest usuwane, usuwane też są wszystkie zadania zależne od niego.

14.9 Zadania i dyskryminanty

W Adzie do określenia bardzo wielu obiektów można stosować parametry zwane też dyskryminantami (ponieważ parametry są zmiennymi przesyłanymi z programu wywołującego do procedury, a dyskryminanty są traktowane jako stałe).

Dyskryminanty używane są nie tylko dla przekazywania zadaniu danych początkowych, ale także do nadawania mu priorytetu, wielkości pamięci zarezerwowanej dla tego zadania, rozmiaru rodzin wejść. Celem takiej konstrukcji jest unikanie konieczności tworzenia spotkania w części inicjacyjnej zadania, co nie tylko zmniejszyłoby czytelność programu, ale także spowodowałoby znaczne natężenie komunikacji w czasie równoległej aktywacji wielu zadań i tym samym zmniejszenie płynności działania całego programu.

Istnieją ograniczenia dotyczące typów dyskryminantów (rozdział 11.1), jednakże bardziej złożone struktury można przesyłać w postaci wskaźników. Można m. in. zawrzeć opis zadania w zbiorze danych tworząc struktury odnoszące się do samych siebie (rozdział 13.3.1):

```
type Zadanie_I_Dane is limited
  record
    ... – pewne dane
    Wykonawca: Pracownik(Zadanie_I_Dane'Access);
          -- Pracownik to typ zadaniowy
  end record;
```

lub

```
type Zasob is
  record
    Licznik: Integer;
    ...
  end record;

protected type Straznik(R: access Zasob) is
  procedure Zwieksz;
  ...
end Straznik;
```

```
protected body Straznik is
  procedure Zwieksz is
  begin
    R.Licznik := R.Licznik + 1;
  end Zwieksz;
  ...
end Straznik;
```

i w treści procedur monitora (np. Zwieksz) możliwy jest bezpieczny dostęp do danych typu Zasob. Poszczególny monitor można zadeklarować jako:

```
Moj_Zasob: aliased Zasob := ...
...
Moj_Obiekt: Straznik(Moj_Zasob'Access);
...
Moj_Obiekt.Zwieksz;
```

14.9.1 Dyskryminanty zadań a programowanie obiektowe

Na pierwszy rzut oka wydaje się, że programowanie obiektowe i zadania są koncepcjami nie mającymi ze sobą nic wspólnego, ponieważ zadań i monitorów nie można rozszerzać np. o dodatkowe wejścia, bariery czy procedury. Jednak w niniejszym rozdziale pokazane będą konstrukcje z własnościami rozszerzania i synchronizacji.

Na przykład zadanie lub monitor może być składnikiem rozszerzalnego obiektu lub odwrotnie, może zawierać taki obiekt. Dającą szerokie możliwości konstrukcją jest konstrukcja zadania lub monitora mająca klasowy dyskryminant (konstrukcja T'Class, rozdział 12).

Pierwszy przykład demonstruje, w jaki sposób typ zadaniowy może stworzyć wzorzec dla zbliżonych operacji, gdzie wywołania procedur zostaną dobrane do aktualnego typu obiektowego przekazywanego jako dyskryminant. Podobnie będzie ze zgłaszaniem wyjątków.

```
1  task type T(Wykonawca: access Opis_Wykonawcy'Class);
2
3  task body T is
4  begin
5    Zacznij(Wykonawca);
6    for I in 1 .. Iteracje (Wykonawca) loop
7      delay Interwal (Wykonawca);
8      Zrob_To (Wykonawca, I);
9    end loop;
10   Zakoncz(Wykonawca);
11   exception
12     when Zdarzenie: others =>
13       Obsluz_Awarie (Wykonawca, Zdarzenie);
14 end T;
```


Trzeba tu podkreślić, że dyskryminant wskaźnikowy Wykonawca jest typem klasowym i w związku z tym odpowiednie procedury (Zacznij, Iteracje, Interwal,

Zrob_To, Zakoncz i Obsluz_Awarie) będą dobrane do aktualnego typu zmiennej Wykonawca.all. Poniżej znajduje się przykład obiektowego typu bazowego, o który oparty jest typ zadaniowy T:

```
package Prosty_Wykonawca is
   type Opis_Wykonawcy is abstract tagged null record;
   procedure Zacznij(J: access Opis_Wykonawcy);
   function Iteracje(J: access Opis_Wykonawcy) return Integer
         is abstract;
   function Interwal(J: access Opis_Wykonawcy) return Duration
         is abstract;
   procedure Zrob_To(J: access Opis_Wykonawcy; I: Integer)
         is abstract;
   procedure Zakoncz(J: access Opis_Wykonawcy);
   procedure Obsluz_Awarie(J: access Opis_Wykonawcy;
                           E: Exception_Occurrence);
end Prosty_Wykonawca;
```

Większość operacji została zdefiniowana jako operacje abstrakcyjne po to, by zmusić użytkownika do dostarczenia wersji nieabstrakcyjnych. Procedury Zacznij i Zakoncz są po prostu puste, standardowa jest także procedura Obsluz_Awarie.

Definicja typu aktualnego bazującego na typie Prosty_Wykonawca może być następująca:

```
with Prosty_Wykonawca; use Prosty_Wykonawca;
package Demonstracja is
   type Demo is new Opis_Wykonawcy with null record;
   function Iteracje(D: access Demo) return Integer;
   function Interwal(D: access Demo) return Duration;
   procedure Zrob_To(D: access Demo; I: Integer);
end;

package body Demonstracja is
   function Iteracje(D: access Demo) return Integer is
   begin
      return 10;
   end Iteracje;

   function Interwal(D: access Demo) return Duration is
   begin
      return 60.0;
   end Interwal;

   procedure Zrob_To(D: access Demo; I: Integer) is
   begin
      New_Line; Put("Minela kolejna sekunda "); Put(I);
   end Zrob_To;

end Demonstracja;
...
Demo: Demonstracja.Demo; -- Dane dla demonstracji
Zadanie_Demonstracyjne: T(Demo'Access); -- Utworzenie zadania
```

Obiekt ten powoduje, że co minutę wypisywany na ekranie jest nowy napis.

Przykład ten jest niezbyt interesujący, ze względu na to, że typ rozszerzony nie zawiera żadnych danych. Poniżej pokazano bardziej złożony przykład wykorzystania bardzo ogólnego zadania Base_Job. Procedura Start może np. sprawdzić, czy demonstracja nie ma trwać zbyt długo, a jeśli tak zgłosi odpowiedni wyjątek.

```
1  package Lepsza_Demonstracja is
2    type Lepsze_Demo is new Opis_Wykonawcy with
3      record
4        Liczba_Iteracji : Integer;
5        Interwal : Duration;
6      end record;
7    Niezbyt_Madre_Demo: exception;
8    ...
9  end;
10
11 package body Lepsza_Demonstracja is
12
13   function Iteracje(D: access Lepsze_Demo) return Integer is
14   begin
15     return D.Liczba_Iteracji;
16   end Iteracje;
17   ...
18   procedure Zacznij(D: access Lepsze_Demo) is
19   begin
20     if D.Liczba_Iteracji * D.Interwał > 300.0 then
21       Raise_Exception (Niezbyt_Madre_Demo'Identity,
22                        "O nie. Za dlugo");
23     end if;
24   end Zacznij;
25
26   procedure Obsluz_Awarie (D: access Lepsze_Demo;
27                            E: Exception_Occurrence) is
28   begin
29     Put_Line("Demonstracja nie udala sie z powodu: ");
30     Put_Line(Exception_Message(E));
31   end Obsluz_Awarie;
32
33 end Lepsza_Demonstracja;
```

W celach ilustracyjnych wykorzystano mechanizm identyfikacji wyjątków opisany w rozdziale 10.6).

Kolejny przykład pokazuje jak typy klasowe mogą być zawarte w monitorze.

Niech pewna kolejka będzie określona następującym obiektem

```
type Kolejka is abstract tagged null record;
function Jest_Pusta(Q: in Kolejka) return Boolean is abstract;
function Jest_Pelna(Q: in Kolejka) return Boolean is abstract;
procedure Dodaj_Do_Kolejki(Q: access Kolejka;
                           X: Odpowiednie_Dane) is abstract;
procedure Usun_Z_Kolejki(Q: access Kolejka;
                         X: out Odpowiednie_Dane) is abstract;
```


Definicja ta jest niezwykle ogólna i odpowiada kolejce zaimplementowanej w dowolny sposób. Konkretna implementacja musi dostarczyć treść tych podprogra-

mów. W szczególności funkcje Jest_Pusta i Jest_Pelna mają parametry w trybie in, ponieważ nie modyfikują one kolejki, natomiast Dodaj_Do_Kolejki i Usun_Z_Kolejki mają parametry wskaźnikowe, ponieważ zmieniają kolejkę. Odpowiedni wzorzec monitora może być zdefiniowany następująco:

```
1  protected type PQ(Q: access Kolejka'Class) is
2     entry Wstaw(X: in Odpowiednie_Dane);
3     entry Weź(X: out Odpowiednie_Dane);
4  end;
5
6  protected body PQ is
7
8     entry Wstaw(X: in Odpowiednie_Dane) when not Jest_Pelna(Q.all) is
9     begin
10       Dodaj_Do_Kolejki(Q, X);
11    end Wstaw;
12
13    entry Wez(X: out Odpowiednie_Dane) when not Jest_Pusta(Q.all) is
14    begin
15       Usuń_Z_Kolejki(Q, X);
16    end Wez;
17 end PQ;
```

Spójność kolejki jest zapewniona w sposób naturalny przez mechanizm monitora. Funkcje Jest_Pusta i Jest_Pelna są używane do obliczania barier. Każda szczególna implementacja jest określona przez dyskryminant wskaźnikowy.

Szczególna implementacja takiej kolejki może być następująca:

```
type Moja_Kolejka is new Kolejka with private;
function Jest_Pusta(Q: Moja_Kolejka) return Boolean;
...
```

a nastepnie zadeklarować kolejkę chronioną monitorem i użyć jej w następujący sposób:

```
Pewna_Kolejka: aliased Moja_Kolejka;
Moja_Kolejka_Zabezpieczona_Monitorem: PQ(Pewna_Kolejka'Access);
...
Moja_Kolejka_Zabezpieczona_Monitorem.Wstaw(Pewien_Element);
```

14.10 Inne przykłady synchronizacji

Poniższy przykład demonstruje deklarację i treść zadania producenta i konsumenta:

```
1  task Producent;
2  task body Producent is
3     Znak : Character;
4  begin
5     loop
6        ... -- produkuje kolejny znak "Znak"
7        Bufor.Pisz(Znak);
8        exit when Znak = ASCII.EOT;
```

```
9    end loop;
10 end Producent;
11
12 task Konsument;
13 task body Konsument is
14    Znak : Character;
15 begin
16    loop
17       Bufor.Czytaj(Znak);
18       exit when Znak = ASCII.EOT;
19       ... -- konsument znaku "Znak"
20    end loop;
21 end Konsument;
```

Bufor zawiera wewnętrzną pulę znaków zarządzana zgodnie z algorytmem karuzeli (ang. *round-robin*). Pula ma dwie wartości charakterystyczne – Indeks_Wejsciowy określający położenie następnego znaku wejściowego i Indeks_Wyjsciowy określający położenie następnego znaku wyjściowego.

```
1  protected Bufor is
2     entry Czytaj (C : out Character);
3     entry Pisz (C : in Character);
4  private
5     Pula : String(1 .. 100);
6     Licznik : Natural := 0;
7     Indeks_Wejściowy, Indeks_Wyjściowy: Positive := 1;
8  end Bufor;
9
10 protected body Bufor is
11
12    entry Pisz(C : in Character) when Licznik < Pula'Length is
13    begin
14       Pula(Indeks_Wejściowy) := C;
15       Indeks_Wejsciowy := (Indeks_Wejsciowy mod Pula'Length) + 1;
16       Licznik := Licznik + 1;
17    end Write;
18
19    entry Read(C : out Character) when Licznik > 0 is
20    begin
21       C := Pula(Indeks_Wyjściowy);
22       Indeks_Wyjsciowy := (Indeks_Wyjsciowy mod Pula'Length) + 1;
23       Licznik := Licznik - 1;
24    end Read;
25
26 end Buffer;
```

14

Więcej informacji nt. technik programowania w języku Ada znaleźć można m.in. w (Mottet i Szmuc 2002).

14.11 Instrukcja `requeue`

W języku Ada możliwa jest także implementacja bardziej złożonych zachowań zadań, związanych z bardzo niedocenianą i mało przejrzyście opisywaną instrukcją requeue opisaną w notacji EBNF jako:

```
requeue Nazwa_Wejscia [with abort];
```

Instrukcja ta może być użyta wyłącznie w obsłudze wejścia, albo wewnątrz instrukcji accept, a jej skutkiem jest przekierowanie zadania do innego (albo tego samego) wejścia danego obiektu chronionego. Ma się rozumieć wejście to może być zablokowane na pewnym warunku, i tym samym zadanie to może zostać zawieszone.

Należy tu odróżnić dwie sytuacje – requeue w wersji wewnętrznej – tj, w takim przypadku, w którym przekierowanie następuje do innego wejścia tego samego obiektu chronionego (monitora) i w wersji zewnętrznej, w którym wywoływane jest wejście innego monitora.

Żeby zrozumieć różnicę trzeba sobie uświadomić jaka jest różnica pomiędzy wywołaniem pewnego wejścia, a następnie wywołaniem innego wejścia tego samego monitora, a wywołaniem wejscia z przekierowaniem. W pierwszym przypadku **nie** jest zagwarantowana spójność monitora, tzn. zmienne ustawione przy pierwszym wywołaniu **nie muszą** być takie same przy kolejnym wywołaniu, ponieważ inne zadanie mogło nam „zepsuć" monitor. W przypadku kiedy następuje przekierowanie istnieje gwarancja, że o ile wejście jest odblokowane – zadanie wywołujące instrukcję będzie kontynuowane nawet, jeżeli ktoś już czeka w kolejce do wywołania tego zadania.

Trzeba tu jescze dodać co oznacza opcjonalne with abort w konstrukcji requeue. Otóż jak Czytelnik lub Czytelniczka pamiętają, instrukcja abort nie usuwa zadania natychmiast, ale m.in. po opuszczeniu monitora. Zatem instrukcja requeue Nazwa_Wejscia spowoduje, że mimo wykonania instrukcji abort dla danego wejścia będzie wykonywała Nazwa_Wejscia, natomiast w przypadku zastosowania instrukcji requeue Nazwa_Wejscia with abort przekierowanie nie nastąpi.

Ponadto, (a jest to bardziej interesujący przypadek) jeżeli z wywołaniem danego wejscia zwiazane są prztermino wania, to modyfikator with abort spowoduje, że dane wejście zostanie uznane za przeterminowane.

Po co taka konstrukcja? Zilustrujemy jej działanie na przykładzie monitora wywołującego odpowiednie wejście w zależności od parametrów formalnych.

```
1  with Ada.Text_IO; use Ada.Text_IO;
2
3  procedure Ins_Requeue is
4
5     protected I is
6        entry E1 (X, Y : Integer);
7        entry E2 (X, Y : Integer);
8        entry E3;
9     private
10       E1_X, E1_Y,
11       E2_X, E2_Y : Integer;
12    end I;
```

```
13
14      protected body I is
15         entry E1 (X, Y : Integer) when True is
16         begin
17            E1_X := X;
18            E1_Y := Y;
19         end E1;
20
21         entry E2 (X, Y : Integer) when True is
22         begin
23            E2_X := X;
24            E2_Y := Y;
25            requeue E3 with abort;
26         end E2;
27
28         entry E3 when E1_X <= E2_X and E1_Y <= E2_Y is
29         begin
30            Put ('*');
31         end E3;
32      end I;
33
34   begin
35      I.E1 (2, 2);
36      for x in 1 .. 4 loop
37         for y in 1 .. 4 loop
38            select
39               I.E2 (x, y);
40            else
41               Put ('.');
42            end select;
43         end loop;
44         New_Line;
45      end loop;
46   end Ins_Requeue;
```

Wynik działania programu jest następujący:

```
....
.***
.***
.***
```

Gwiazdki «*» pokazują przyjęte wejście, natomiast kropki «.» pokazują wejście zablokowane przekazywanymi do niego parametrami formalnymi.

Poniżej przedstawiamy niepełny przykład konstrukcji, w której możliwe jest rekurencyjne wywołanie wejść przez pewien proces, blokując tymczasem inne procesy w tym monitorze (konstrukcję taką nazywamy *semaforem uogólnionym*).

```
1  entry Get when True is
2     use type Ada.Task_Identification.Task_Id;
3  begin
4     if Use_Count = 0 then
5        Owner_Task_Id := Get'Caller;
```

```
6           Use_Count := 1;
7       elsif Owner_Task_Id = Get'Caller then
8           Use_Count := Use_Count + 1;
9       else
10          requeue Wait_for_Free with abort;
11                  -- Jeżeli przeterminowania są aktywne,
12                  -- to niech wystąpią.
13      end if;
14  end Get;
15
16  entry Wait_for_Free when Use_Count = 0 is
17  begin
18      Owner_Task_Id := Wait_for_Free'Caller;
19      Use_Count := 1;
20  end Wait_for_Free;
21
22  procedure Release is
23      use type Ada.Task_Identification.Task_Id;
24  begin
25      if Release'Caller /= Owner_Task_Id then
26          raise Use_error;
27      else
28          Use_Count := Use_Count - 1;
29          if Use_Count = 0 then
30              Owner_Task_Id := Null_task_ID;
31          end if;
32      end if;
33  end Release;
```

Czy Czytelniczka/Czytelnik wie dlaczego stosujemy konstrukcję «when True»? Jeśli nie, to przypomnimy, że przekierować wywołanie można tylko pomiędzy wejściami monitora. A, że nie ma sensu blokować wejścia Get – trzeba właśnie napisać warunek «when True».

14.12 Ćwiczenia

◁ Ćwiczenie 14.1 ▷▷▷ Napisz program, w którym w oknie poruszają się „duszki", z których każdy kontrolowany jest przez osobny proces.

◁ Ćwiczenie 14.2 ▷▷▷ Zmodyfikuj powyższy program w taki sposób, że w danym oknie powstają aktywne obszary takie, że w jeżeli jeden z „duszków" wpadnie w obszar pierwszy, to powstaje gdzieś nowy proces, a jeżeli wpadnie w drugi obszar, to proces odpowiedzialny za tego „duszka" umiera. **Pamiętaj, że nie można zakończyć procesu, który uruchomił inne procesy** i dlatego wszystkie procesy powinny mieć wspólnego rodzica.

◁ Ćwiczenie 14.3 ▷▷▷ Czy twój program da się *łagodnie* zakończyć np. poprzez wciśnięcie jakiegoś klawisza? A jeśli nie, to zmodyfikuj go odpowiednio.

◁ Ćwiczenie 14.4 ▷▷▷ Napisz program, który czyta znaki z portu szeregowego, do którego przyłączona jest myszka. Należy zinterpretować znaki generowane przez sprzęt (odgadnąć protokół komunikacji) i na ich podstawie rysować na ekranie własny kursor.

◁ Ćwiczenie 14.5 ▷▷▷ Połącz oba te programy w taki sposób, żeby za pomocą myszki można było wstrzymywać procesy sterujące „duszkami" (jeżeli kusor myszki znajdzie się w pobliżu „duszka"). Jednakże pamiętaj, że procesy te nie mogą oczekiwać aktywnie – tzn. nie mogą okresowo sprawdzać stanu myszki.

◁ Ćwiczenie 14.6 ▷▷▷ Zmodyfikuj program w taki sposób, żeby proces sterujący każdym „duszkiem" ciągnął za soba wygasający ogon, który można zinterpretować jako smugę kondensacyjną za samolotem. Smuga taka z czasem rozprasza się i ginie i w związku z tym jest dłuższa jeżeli proces porusza „duszkiem" szybciej, krótsza – jeżeli porusza się wolniej i wcale jej nie ma – jeśli „duszek" stoi.

◁ Ćwiczenie 14.7 ▷▷▷ Napisz program, w którym N procesów jest wznawianych synchronicznie przez jedno wyróżnione zadanie. Należy zwrócić uwagę na to, że każde zadanie musi być uruchomione za każdym razem tylko jednokrotnie. Problem rozwiązać za pomocą monitorów i za pomocą spotkań.

◁ Ćwiczenie 14.8 ▷▷▷ Napisz program, w którym dwa procesy–producenci produkują asynchronicznie dwie zmienne co w przybliżeniu taki sam okres. Proces konsument ma przeczytać obie zmienne po tym jak obie się zmienią. Proces producent nie może wytworzyć nowej zmiennej dopóki konsument jej nie przeczyta.

◁ Ćwiczenie 14.9 ▷▷▷ Co trzeba zmienić w programie, jeżeli istnieje wiele programów – konsumentów?

Dodatek A

Raport NIST

28 czerwca 2002 roku, agencja Reutera podała przedstawioną poniżej informację, przy czym sugerujemy zapoznanie się szczególnie z jej ostatnim zdaniem. Język Ada, a w szczególności język Ada95 jest tak zaprojektowany, że powstawanie *głupich* błędów programistycznych jest w nim utrudnione. „Namówienie" kompilatora do zrobienia jakieś operacji niezgodnej z ogólnymi zasadami tworzenia oprogramowania (np. operacje na wskaźnikach) jest bardzo męczące i zniechęca programistę do robienia takich „skrótów", co ma ogólnie bardzo pozytywny wpływ na jakość tworzonego oprogramowania.

Treść raportu

Według badań amerykańskiego oddziału NIST[1] błędy i niewłaściwe działanie oprogramowania kosztuje amerykańską gospodarkę około 59.5 miliarda dolarów rocznie. Dyrektor NIST Arden Barment powiedział na konferencji prasowej, że „wpływ błędów w oprogramowaniu jest olbrzymi, ponieważ w zasadzie każdy rodzaj działalności gospodarczej w Stanach Zjednoczonych jest zależny od oprogamowania we wszystkich swoich aspektach, takich jak projektowanie, produkcja, dystrybucja, sprzedaż, serwis, usługi posprzedażne".

Mniej więcej połowa problemu leży po stronie użytkowników oprogramowania, ale niestety jego wytwórcy i sprzedawcy są odpowiedzialni za resztę. NIST twierdzi, że lepsze testowanie oprogramowania spowodowałoby wykrywanie błędów umożliwiając ich usuwanie we wczesnym okresie projektowania, co mogłoby spowodowac redukcję kosztów o ok. 22.2 miliarda dolarów. Obecnie około połowy wszystkich błędów nie zostaje znalezione do momentu wprowadzenia oprogramowania do sprzedaży.

[1] **N**ational **I**nstitute of **S**tandards and **T**echnology

Badania prowadzone przez Research Triangle Institute in North Carolina i przemysł *software'owy* wskazały na techniczną potrzebę zintensyfikowania procesu testowania oprogramowania i wskazały odpowiednie metody.

Oprogramowanie jest podatne na błędy, częściowo ze względu na stopień złożoności milionów linii kodu. Około 80% kosztów tworzenia oprogramowania jest związane z identyfikacją i korekcją błędów. Inne czynniki wpływające na ten problem, to strategie marketingowe, ograniczone możliwości sprzedawców, zmniejszający się czas testowania i uruchamiania.

Poza oprogramowaniem tylko nieliczne produkty charakteryzują się zbliżonym poziomem błędów.

W styczniu 2002 roku NAS[2] napisała raport żądający, by ustawodawcy **pilnie** rozważyli stworzenie prawa umożliwiającego dochodzenie roszczeń wynikłych ze strat spowodowanych niewłaściwym działaniem oprogramowania.

Jeżeli twórcy oprogramowania nie będą się czuli odpowiedzialni za działanie swojego oprogramowania – koszt ich błędów wzrośnie w najbliższym czasie dramatycznie (ale koszt ten zapłacą klienci).

W Europie tak już się dzieje. We wrześniu 2001 roku holenderski sędzia ukarał przedsiębiorstwo Exact Holding za złą praktykę sprzedawania oprogramowania zawierającego błędy, odrzucając argument, że wczesne wersje oprogramowania są tradycyjnie niestabilne.

[2] **N**ational **A**cademy of **S**ciences

Bibliografia

Alagić, S. i Arbib, M. (1978), *The design of well-structured and correct programs*, Springer, New York. Istnieje polskie tłumaczenie.

Arbib, M., Kfoury, A. i Moll, R. (1981), *A basis for theoretical computer science*, Springer, New York.

Arendt, D., Postół, M. i Zajączkowski, A. (1988), *Modula–2*, NOT, Warszawa.

Barnes, J. (1998), *Programming in Ada 95*, Addison-Wesley, Harlow, England.

Bazaraa, M. i Shetty, C. (1982), *Nieliniejnoje programirowanije. Tieorija i algoritmy*, Mir, Moskwa. tłum. z ang.

Beidler, J. (1997), *Data structures and Algorithms. An object-oriented approach using Ada 95*, Springer, New York.

Ben-Ari, M. (1996), *Podstawy programowania współbieżnego i rozproszonego*, WNT, Warszawa. tłum. z ang.

Birkhoff, G. i Bartee, T. (1983), *Współczesna algebra stosowana*, PWN, Warszawa. tłum. z ang.

Buchwald, M. (2001), Program do automatycznego tłumaczenia programów z języka Delphi na język Ada95, Technical report, Samodzielny Zakład Sieci Komputerowych. Magisterska praca dyplomowa napisana pod kierunkiem dr inż. Michała Morawskiego.
URL ⇒ `http://zskl.zsk.p.lodz.pl/~morawski/dyplomy/`

Coad, P. i Nicola, J. (1993), *Programowanie obiektowe*, Oficyna wydawnicza READ ME, Warszawa. tłum. z ang.

Cooling, J. (1996), 'Languages for the programming of real-time embedded systems. a survey and comparison', *Microprocessors and Microsystems* **20**, 67–77.

Cormen, T., Leiserson, C. i Rivest, R. (1997), *Wprowadzenie do algorytmów*, WNT, Warszawa. tłum. z ang.

Dale, N., Weems, C. i McCormick, J. (2000), *Programming and problem solving with Ada95*, wydanie drugie, Jones and Bartlett publishers, Inc.

Eckel, B. (2001), *Thinking in Java, edycja polska*, Wydawnictwo Helion, Polska. Wersja angielska dostępna jest w postaci elektronicznej pod adresem `http://www.bruceeckel.com`.

Green, R. (2001), 'How to write unmaintainable code', Strona WWW.
URL ⇒ `http://www.freevbcode.com/ShowCode.Asp?ID=2547`

Habermann, A. i Perry, D. (1989), *Ada dla zaawansowanych*, WNT, Warszawa. tłum. z ang.

Harel, D. (1992), *Rzecz o istocie informatyki. Algorytmika*, WNT, Warszawa. tłum. z ang.

Huzar, Z., Fryźlewicz, Z., Dubielewicz, I., Hnatkowska, B. i Waniczek, J. (1998), *Ada 95*, Helion, Gliwice.

Intermetrics Inc., Cambridge, M. (1995*a*), 'Ada 95 rational', Strona WWW.
URL ⇒ `ftp://ftp.cs.nyu.edu/pub/gnat/rationale-ada95/`

Intermetrics Inc., Cambridge, M. (1995*b*), 'Ada 95 reference manual', Strona WWW.
URL ⇒ `ftp://ftp.cs.nyu.edu/pub/gnat/rm9x-v5.95/`

Kernighan, B. i Ritchie, D. (1987), *Język C*, WNT, Warszawa.

Kopetz, H. (1998), *Real-Time Systems, Design Principles for Distributed Embedded Applications*, wydanie drugie, Kluwer Academic Publishers.

Marcotty, M. i Ledgard, H. (1991), *W kręgu języków programowania*, WNT, Warszawa. tłum. z ang.

Martin, J. i Odell, J. (1997), *Podstawy metod obiektowych*, WNT, Warszawa. tłum. z ang.

Małecki, R., Arendt, D., Bryszewski, A. i Krasiukianis, R. (1997), *Wstęp do informatyki*, Wydawnictwo Politechniki Łódzkiej, Łódź.

Mottet, G. i Szmuc, T. (2002), *Programowanie systemów czasu rzeczywistego z zastosowaniem języka Ada*, Uczelniane Wydawnictwa Naukowo-Dydaktyczne, Kraków.

Oktaba, H. i Ratajczak, W. (1980), *Simula-67*, WNT, Warszawa.

Papoulis, A. (1988), *Obwody i układy*, WKŁ, Warszawa. tłum. z ang.

Petzold, C. i Yao, P. (1997), *Programowanie Windows 95*, Oficyna Wydawnicza READ ME, Warszawa. tłum. z ang.

Polak, E. (1971), *Computational methods in optimization*, Academic Press, New York.

Pyle, I. (1986), *Ada*, WNT, Warszawa. tłum. z ang.

Ross, K. i Wright, C. (1999), *Matematyka dyskretna*, PWN, Warszawa. tłum. z ang.

Schwarz, M. i Shaw, L. (1975), *Signal processing*, McGraw Hill, New York.

Smith, M. (1996), *Object-oriented software in Ada 95*, International Thomson Computer Press, London.

Stroustrup, B. (1991), *The C++ Programming Language*, wydanie drugie, Addison-Wesley Publishing Company.

Teixeira, S. i Pacheco, X. (2002), *Delphi 6. Vademecum profesjonalisty.*, Helion.

Trajdos, T. (1998), *Matematyka, część III*, wydanie dziewiąte, WNT, Warszawa.

van Tassel, D. (1982), *Praktyka programowania*, WNT, Warszawa. tłum. z ang.

Wiener, R. i Sincovec, R. (1984), *Software engineering with Modula 2 and Ada*, John Wiley & Sons, New York.

Wirth, N. (1978), *Wstęp do programowania systematycznego*, WNT, Warszawa. tłum. z niem.

Wirth, N. (1989), *Algorytmy + struktury danych = programy*, wydanie drugie, WNT, Warszawa. tłum. z ang.

www.rtj.org (2001), 'Real time java specification', Strona WWW.
URL ⇒ `http://www.rtj.org`

www.unicode.org (2000), 'Standard unicode', Strona WWW.
URL ⇒ `http://www.unicode.org`

Żakowski, W. i Decewicz, G. (1997), *Matematyka, część I*, wydanie piętnaste, WNT, Warszawa.

Skorowidz