Politechnika Śląska
Wydział Automatyki, Elektroniki i Informatyki
Instytut Informatyki

Michał Cierniak

Opracowanie i wykonanie modułu kompilatora języka ADA systemu IIPS/Ada generującego kod wynikowy w postaci P-kodu

PRACA MAGISTERSKA

Promotor:

dr inż. Przemysław Szmal

Gliwice, 1990

Spis treści.

Wstęp. 3

1. Miejsce generatora kodu pośredniego w procesie kompilacji. 5

2. DIANA - postać pośrednia programów zapisanych w Adzie. . . 7
2.1. Informacje ogólne 7
2.2. Opis języka DIANA. 7
2.3. Postać zewnętrzna języka DIANA. 9

3. A-kod - wersja P-kodu dostosowana do języka Ada. 10
3.1. P-kod jako język pośredni dostosowany do maszyny realizującej programy napisane w Pascalu. 10
3.2. Rozszerzenia A-kodu w stosunku do oryginalnego P-kodu. 10
3.3. Kompilowanie deklaracji. 12
3.4. Kompilowanie wyrażeń. 14
3.5. Kompilowanie instrukcji. 18
3.5.1. Podstawienie. 18
3.5.2. Instrukcja warunkowa. 19
3.5.3. Instrukcja wyboru. 20
3.5.4. Instrukcje pętli. 21
3.5.5. Instrukcja bloku. 22
3.5.6. Instrukcja wyjścia z pętli. 22
3.5.7. Instrukcja powrotu z podprogramu. 23
3.5.8. Instrukcja skoku. 23
3.6. Kompilowanie procedur i funkcji. 24
3.6.1. Kompilowanie wywołań procedur i funkcji. 24
3.6.2. Kompilowanie deklaracji procedur. 25
3.7. Rozłączna kompilacja. 28
3.7.1. Pakiet jako jednostka kompilacji. 29
3.7.2. Treść pakietu jako jednostka kompilacji. 29
3.7.3. Procedura jako jednostka kompilacji. 31
3.8. Kompilowanie zadań. 32
3.8.1. Deklaracje. 32
3.8.2. Wejścia (instrukcje accept, select). 34
3.8.3. Wywołanie wejścia. 37
3.8.4. Instrukcja abort. 38
3.8.5. Atrybuty zadań. 39
3.9. Kompilowanie wyjątków. 39

4. Opis implementacji. 41
4.1. Wykorzystywane struktury danych. 41
4.2. Struktura wykonanego modułu generatora kodu pośredniego. 42

5. Opis działania programu. 47

6. Przebieg prac. 48

7. Podsumowanie. 50

Dodatek 1. Definicja języka DIANA. 52

Dodatek 2. Instrukcje A-kodu. 68

Dodatek 3. Standardowe procedury A-kodu. 76

Dodatek 4. Zapis pakietu STANDARD w postaci DIANY. 78

Dodatek 5. Wybrane przebiegi testowania. 82
Bibliografia 97

Wstęp.

Tematem mojej pracy było opracowanie modułu generatora kodu kompilatora języka Ada. Zadanie to poza częścią wstępną, polegającą na zapoznaniu się z konstrukcją kompilatorów i językiem Ada, można podzielić na dwa etapy. Pierwszym było zaprojektowanie takiej wersji P-kodu, która umożliwiałaby wykonywanie programów napisanych w języku Ada oraz szczegółowe ustalenie sposobu tłumaczenia na tak rozszerzony P-kod (który nazwałem **A-kodem**) poszczególnych konstrukcji języka Ada. Efektem tej części było opracowanie szczegółowej specyfikacji dla konstruowanego przeze mnie modułu kompilatora. Częścią drugą było zaimplementowanie modułu generatora kodu jako programu tłumaczącego jednostkę kompilacji zapisaną w ustalonej postaci pośredniej na wcześniej zaprojektowany A-kod. Jako postać pośrednią Ady, generowaną przez moduł analizy semantycznej, wybraliśmy język **DIANA**. DIANA jest dość szeroko wykorzystywana w już istniejących kompilatorach i innych narzędziach będących częścią środowiska programowego Ady.

Ze względu na złożoność języka Ada musiałem zdecydować się na implementowanie tylko pewnego podzbioru języka. W części pierwszej (projektowej) nie rozpatrywane przeze mnie elementy Ady to: podjednostki, typy rzeczywiste i mechanizmy niskiego poziomu. Zdecydowałem się na pominięcie tych trzech elementów, gdyż moim zdaniem nie wnoszą one nic nowego do problematyki kompilacji, a uwzględnienie ich spowodowałoby znaczny wzrost złożoności mojej pracy. W części drugiej (praktycznej) zdecydowałem się na dalsze ograniczenia.

Rozdział 1 mojej pracy zawiera ogólne omówienie problematyki kompilacji oraz funkcji generatora kodu w kompilatorze. Omówione są także wady i zalety translatorów wykorzystujących technikę kompilacji z generacją kodu pośredniego.

W **Rozdziale 2** znajduje się opis DIANY - języka pośredniej reprezentacji programów napisanych w Adzie. Uzupełnieniem tego rozdziału jest **Dodatek 1** zawierający opis DIANY z wykorzystaniem notacji IDL.

Rozdział 3 zawiera opis A-kodu i sposobu tłumaczenia poszczególnych konstrukcji Ady na A-kod. Rozdział ten jest

uzupełniony Dodatkiem 2 i Dodatkiem 3. Dodatki te zawierają alfabetyczną listę (wraz z krótkim opisem) instrukcji i procedur standardowych A-kodu.

W Rozdziale 4, poświęconym opisowi wykonanego przeze mnie modułu, znajduje się opis wykorzystywanych struktur danych oraz opis struktury modułu generatora kodu pośredniego. Jako uzupełnienie tego rozdziału można traktować Dodatek 4 zawierający pakiet STANDARD przedstawiony w języku DIANA.

Rozdział 5 zawiera opis działania i wykorzystania wykonanego modułu.

Tematyką Rozdziału 6 jest przebieg prac projektowych i praktycznych prowadzących do wykonania modułu generatora A-kodu. Uzupełnieniem tego rozdziału jest Dodatek 5 zawierający wybrane przebiegi testowania.

Rozdział 7 - ostatni zawiera podsumowanie części projektowej i praktycznej mojej pracy.

1. Miejsce generatora kodu pośredniego w procesie kompilacji.

Kompilator jest programem, który czyta program napisany w jednym języku, zwanym **językiem źródłowym**, i tłumaczy go na równoważny program w drugim języku - **języku docelowym**. Zwykle językiem źródłowym jest język programowania a językiem docelowym - język maszynowy jakiegoś komputera. Tak jest m.in. w przypadku kompilatora Ady, którego częścią jest projektowany przeze mnie moduł.

Kompilację można rozbić na dwa zadania - **analizę i syntezę**. Strukturą danych pośredniczącą między tymi etapami (tzn. tworzoną podczas analizy i stanowiącą dane wejściowe dla syntezy) jest drzewo struktury.

Główną częścią syntezy jest **generacja kodu**. Jak stwierdziłem wcześniej, zadaniem kompilacji jest wyrażenie algorytmu w formie zrozumiałej dla komputera. Często jednak najpierw generowany jest program wyrażony w języku maszynowym abstrakcyjnego komputera, w tzw. **języku pośrednim**. Dopiero ten język pośredni tłumaczony jest na język wewnętrzny komputera docelowego. Głównym powodem, dla którego przyjmowana jest taka strategia, jest zwiększenie **przenośności** kompilatora. Dla przeniesienia kompilatora na inny komputer wystarczy bowiem, w tym przypadku, zmienić tylko te moduły, które zależą od własności języka docelowego. Często kompilator dzieli się na **część przednią** (ang. front end) i **część tylną** (ang. back end). Część przednia zależy od własności języka źródłowego, a część tylna - od własności języka docelowego. Część przednia tworzy więc reprezentację kompilowanego programu w języku pośrednim. Przy takim podziale przeniesienie kompilatora na inny komputer wymaga zmian tylko części tylnej. Przy odpowiednim jej projektowaniu nawet te zmiany mogą być niewielkie.

Innym powodem kompilacji z wykorzystaniem języka pośredniego może być chęć wykorzystania już istniejącego interpretera lub assemblera języka pośredniego.

Podejście takie ma jednak również pewne wady. Powoduje powstanie dodatkowej fazy kompilacji, a ponieważ program wyrażony w języku pośrednim zapisywany jest w pliku - również dodatkowego przebiegu kompilatora. Sprawia to, że tak zaprojektowany kompilator jest wolniejszy w działaniu. Inna wada jest to, że pewne własności tłumaczonego programu, umożliwiające

optymalizację generowanego kodu, mogą być tracone w wyniku tłumaczenia na język pośredni. Wygenerowany program maszynowy może być mniej efektywny niż w przypadku kompilatorów tworzących wprost kod maszynowy.

Ponieważ Ada jest językiem bardzo dużym, sądzę, że wybór rozwiązania z językiem pośrednim jest w naszym wypadku dużo korzystniejszy. Umożliwia bowiem przyszłe uruchomienie kompilatora na komputerze o innej architekturze.

Opis wykorzystywanego przeze mnie języka pośredniego (A-kodu) znajduje się w rozdziale 3 mojej pracy.

2. DIANA - postać pośrednia programów zapisanych w Adzie.

W rozdziale tym omówię stosowaną przeze mnie postać pośrednią dla języka Ada. Pełna jej definicja znajduje się w pracy [1]. Do zrozumienia DIANY (bez pewnych, nieistotnych w mojej pracy, niuansów) wystarczy zapoznanie się z tym rozdziałem oraz z opisem zawartym w Dodatku 1.

2.1. Informacje ogólne

DIANA jest formą postaci pośredniej programów napisanych w języku Ada. Może ona być wykorzystywana przez różne narzędzia stanowiące otoczenie programowe kompilatora języka Ada. Nadaje się również doskonale na język, w którym zapisane są dane wejściowe dla modułu generatora kodu.

W niniejszej implementacji postać pośrednia kompilowanej jednostki kompilacji może być odczytana z pliku zewnętrznego, gdzie jest zapisana w postaci tekstowej. Jako docelowa wydaje się jednak najkorzystniejsza implementacja, w której drzewo odpowiadające opisowi danej jednostki kompilacji jest tworzone przez moduł analizatora semantycznego i znajduje się cały czas (tzn. podczas analizy semantycznej i generacji kodu) w pamięci operacyjnej komputera. Takie rozwiązanie narzucałoby, co prawda, pewne ograniczenia na maksymalny rozmiar jednostki kompilacji, jednak spowodowałoby równocześnie znaczne przyspieszenie procesu kompilacji.

2.2. Opis języka DIANA.

W niniejszej implementacji wykorzystywany jest język DIANA w postaci zdefiniowanej w pracy [1]. Konkretne struktury danych zostały zaprojektowane tak, by jak najlepiej wykorzystać możliwości komputera typu IBM PC i kompilatora Turbo Pascal 5.5.

Definicja DIANY znajduje się w Dodatku 1. Definicja jest przedstawiona z wykorzystaniem zapisu IDL ([7]), opartego na zapisie BNF. Dużymi literami wyróżnione są nazwy klas. Nazwy węzłów i atrybutów pisane są małymi literami.

Np. równoważne zapisy:

```
BLOCK_STUB ::= block;
BLOCK_STUB ::= stub;
 i:
BLOCK_STUB ::= block |
               stub;
```

oznaczaja, że do klasy BLOCK_STUB należa węzły block i stub.

Węzły opisywane są przez swoje atrybuty. Rozróżnia się atrybuty czterech rodzajów. Rodzaje te mogą być rozróżniane na podstawie nazw atrybutów za pomocą następujących dwuliterowych przedrostków:

as_ - atrybuty strukturalne (syntaktyczne),
cd_ - atrybuty tworzone i wykorzystywane przez generator kodu.
lx_ - atrybuty leksykalne,
sm_ - atrybuty semantyczne.

Moduły kompilatora, działające przed generatorem kodu, tworzą postać pośrednia danej kompilacji wypełniając atrybuty leksykalne, składniowe i semantyczne. Np. węzeł:

```
range => lx_srcpos : source_position,
         as_exp1 : EXP,
         as_exp2 : EXP,
         sm_base_type : TYPE_SPEC;
```

posiada jeden atrybut leksykalny lx_srcpos, dwa atrybuty składniowe - as_exp1 i as_exp2 oraz jeden atrybut semantyczny- sm_base_type.

W pracy [1] wykorzystywane sa następujące typy atrybutów:

klasa:
w niniejszej pracy implementowany jest jako typ TREE, mogacy wskazywać na dowolny węzeł.

source_position:
opisuje plik, numer wiersza i numer kolumny, w której znajduje się opisywana konstrukcja w tekscie źródłowym programu.

symbol_rep:
zawiera reprezentację zewnętrzną opisywanego obiektu.

Boolean:
wartosci logiczne TRUE i FALSE.

Integer:
liczba całkowita.

Operator:
typ wyliczeniowy opisujacy operatory dostepne w jezyku Ada.

lista (sequence):
ciag klas.

Rational:

wartość uniweralnego typu rzeczywistego (atrybuty tego typu reprezentują zarówno wartości zmienno- jak i stałopozycyjne).

Value:

wartości obiektów języka Ada (typu logicznego, rzeczywistego, całkowitego lub napisowego).

2.3. Postać zewnętrzna języka DIANA.

Wykorzystywana jest tekstowa postać zapisu języka DIANA zaproponowana w rozdziale 5 pracy [1]. W celu uproszczenia odczytu tak zapisanej kompilacji przyjęto następujące założenia:

- Wykorzystywana jest tylko tzw. płaska postać zapisu (ang. flat form).
- Węzły są reprezentowane przez nazwy składające się z litery 'D' oraz liczby naturalnej. Liczby mniejsze od tysiąca wykorzystywane są dla opisu węzłów pakietu STANDARD. Węzły wszystkich innych jednostek kompilacji (także pakietów standardowych) mają numery większe od tysiąca.
- Każdy atrybut zapisany jest w jednym wierszu.
- Postać zapisu nie zawiera błędów. Wykrywane są tylko niektóre błędy (np. zły typ atrybutu).
- Atrybuty są rozpoznawane na podstawie kolejności występowania, a nie na podstawie nazwy atrybutu. Wykorzystywana jest kolejność podana w dodatku IV pracy [1].
- Atrybut lx_comments nie jest wykorzystywany.
- Atrybut lx_srcpos jest opcjonalny. W dowolnym z węzłów może być podany lub nie. Jeżeli w danym węźle występuje - musi być pierwszym atrybutem.

3. A-kod - wersja P-kodu dostosowana do języka Ada.

W rozdziale tym omówię A-kod. W punkcie 3.1 omówię w skrócie P-kod, na którego podstawie opracowałem wykorzystywany przeze mnie język pośredni. W punkcie 3.2 omówię najważniejsze różnice między P-kodem a zaprojektowanym przeze mnie A-kodem. Punkty 3.3 - 3.8 poświęcone są sposobowi tłumaczenia na A-kod poszczególnych konstrukcji języka Ada.

3.1. P-kod jako język pośredni dostosowany do maszyny realizującej programy napisane w Pascalu.

Jako podstawa do zaprojektowania języka pośredniego w niniejszej inplementacji został wykorzystany P-kod ([3]). P-kod jest językiem maszynowym hipotetycznego komputera, który z wielu powodów jest idealny dla potrzeb języka Pascal. Komputer ten, zwany maszyną P-kodu, jest komputerem zorientowanym stosowo. Oznacza to, że zmienne, parametry, wyniki pośrednie oraz dane organizacyjne umieszczane sa na stosie.

Różnice między Adą a Pascalem powodują jednak, że nie można wykorzystać P-kodu wprost jako języka pośredniego dla Ady. Zaprojektowaną przeze mnie rozszerzoną wersję P-kodu nazwałem A-kodem. Nazwa ta wyraża fakt, że język ten został stworzony tak, by jak najlepiej odpowiadać potrzebom Ady.

3.2. Rozszerzenia A-kodu w stosunku do oryginalnego P-kodu.

3.2.1. Uwagi ogólne.

Zmiany wprowadzone przeze mnie można podzielić na trzy grupy:

- Dodatkowe instrukcje niezbędne dla wykonania programów napisanych w Adzie. Np. instrukcja A-kodu ABO umożliwia tłumaczenie instrukcji abort Ady.
- Zmiany w sposobie działania hipotetycznej maszyny, wynikające ze specyficznych mechanizmów Ady. Np. konieczność istnienia osobnych stosów dla działających współbieżnie zadań.
- Zmiany nie wymuszone przez semantykę Ady, ale wprowadzone, by umożliwić bardziej efektywne tłumaczenie instrukcji A-kodu na instrukcje maszynowe rzeczywistych procesorów. Np. sposób przekazywania parametrów podprogramów.

Następujące instrukcje P-kodu zostały wykorzystane w A-kodzie bez zmian lub ze zmianami zaproponowanymi w pracy [3]:

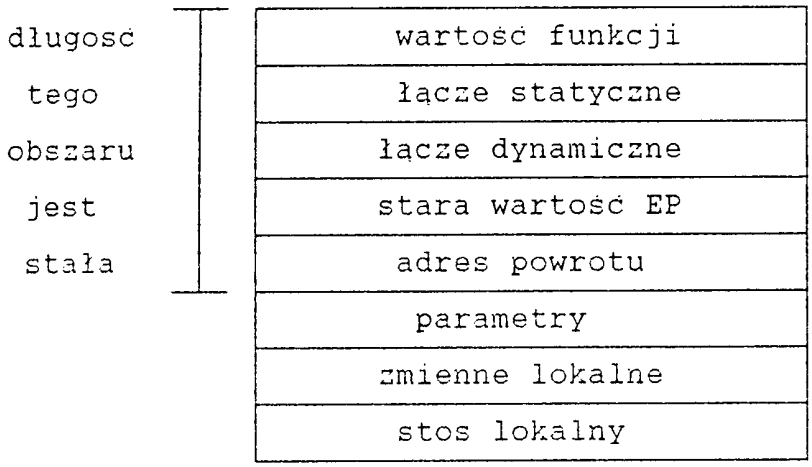
ABS, ADD, CHR, CSP, CUP, DEC, DIV, ENT, EQU, FJP, GEQ, GRE, INC, IND, IXA, LDA, LDC, LEQ, LES, LOD, MOD, MOV, MUL, NEG, NEQ, NOT, STO, STR, SUB, TJP, UJP, XJP.

Wszystkie pozostałe instrukcje zostały bądź zmienione bądź dodane przeze mnie. Alfabetyczny spis wszystkich instrukcji A-kodu, wraz z opisem działania, zamieszczony został w **Dodatku 2**.

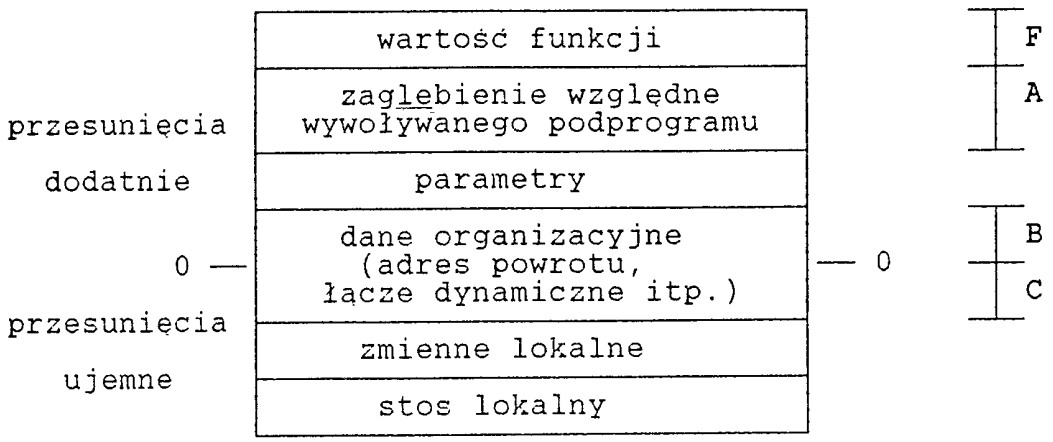
Zaistniała również konieczność opracowania nowych procedur standardowych A-kodu. Dotyczą one głównie operacji na tablicach, zakresach i atrybutów zadań. Pełna ich lista, wraz z opisem działania znajduje się w **Dodatku 3**.

3.2.2. Struktura ramki.

Jedną z najważniejszych zmian w stosunku do P-kodu jest organizacja ramki w maszynie A-kodu. Konieczność tych zmian wynikła zarówno z rozszerzeń Ady w stosunku do Pascala (w Adzie funkcja może zwracać wartość dowolnego typu), jak i z chęci efektywnego wykorzystania instrukcji maszynowych rzeczywistych procesorów (umieszczanie na stosie, za pomocą prostych instrukcji, adresu powrotu i innych danych organizacyjnych).
Ramka w maszynie P-kodu (porównaj [3] s.117) wygląda następująco:



Struktura ramki w maszynie A-kodu:



Wartość F (rozmiar obiektu zwracanego przez funkcję) jest okreslona przez pierwszy parametr instrukcji MST. Dla procedur wartość ta jest zerowa.

Wartości A (długosć przechowywanego zagłebienia względnego wywoływanego podprogramu - drugi parametr instrukcji MST), B (przesunięcie początku obszaru parametrów) oraz C (przesuniecie pierwszej zmiennej lokalnej) określone sa w napisanym przeze mnie module przez stałe (odpowiednio - RelativeResultOffset, FirstParamOffset i FirstLocalVarOffset).

W wypadku zmian w architekturze komputera docelowego wystarczy więc zmienić te stałe.

3.3. Kompilowanie deklaracji.

W punkcie tym omówione są te deklaracje, ktorych kompilacja powoduje wygenerowanie kodu.

3.3.1. Deklaracje typów.

3.3.1.1. Typy tablicowe.

O ile typ jest ograniczony (ang. constrained) inicjalizowana jest struktura opisująca typ tablicowy:

ARRSIZE
IDXFAC
FIRST
LAST

Gdzie: ARRSIZE oznacza rozmiar tablicy w bajtach. Trójka liczb: IDXFAC, FIRST, LAST powtarza się dla każdego wymiaru tablicy i określa:

- IDXFAC - współczynnik skalujący dla operacji indeksowania,
- FIRST - wartość atrybutu 'FIRST,
- LAST - wartość atrybutu 'LAST.

Przykładowo deklaracja:

```
type T is array (1..10, 0..9) of INTEGER;
```

zostaje przetłumaczona na:

```
LDCI   2
STRI   0   -8
LDCI   0
STRI   0   -10
LDCI   9
STRI   0   -12
LDA    0   -10
CSP    LEN
LODI   0   -8
MULI
STRI   0   -2
LDCI   1
STRI   0   -4
LDCI   10
STRI   0   -6
LDA    0   -4
CSP    LEN
LODI   0   -2
MULI
STRI   0    0
```

0: 1:	ARRSIZE
2: 3:	IDXFAC(1)
4: 5:	FIRST(1)
6: 7:	LAST(1)
8: 9:	IDXFAC(2)
10: 11:	FIRST(2)
12: 13:	LAST(2)

Dla tablic o rozmiarach statycznych (znanych na etapie kompilacji) możnaby generowac kod w sposób bardziej optymalny. Spowodoałoby to jednak znaczne skompilkowanie budowy generatora kodu.

3.3.2. Deklaracje obiektów.

3.3.2.1. Zmienne typu tablicowego.

Opis obiektu typu tablicowego jest przechowywany w pamięci jako para wskaźników:

- Wskaźnik do wartości (do przydzielonego obszaru pamięci). Stosowana przeze mnie konwencja zapisu ^wartość oznacza wskaźnik do wartości, analogicznie ^opis oznacza wsaźnik do opisu.
- Wskaźnik do opisu typu.

^wartośc
^opis

W przypadku deklaracji zmiennej z jednoczesną deklaracją typu anonimowego tworzona jest najpierw struktura opisująca ten typ. Zawsze natomiast jest generowany ciąg instrukcji powodujący przydzielenie pamięci dla danej zmiennej oraz zapamiętanie wskaźnika do przydzielonego obszaru pamięci i wskaźnika do opisu typu. Np. deklaracja:

```
 -- type T is array (1..10, 0..9) of INTEGER;
 A: T;
```

zostaje przytłumaczona na:

```
   LDA    0    0          -- pobierz wskaźnik do opisu typu
   DPLA
   STRA   0   -18         -- umieść go pod odpowiednim adresem
   INDI   0               -- pobierz rozmiar tablicy
   ALO    0               -- zażądaj przydzielenia pamięci
   STRA   0   -14         -- i zachowaj uzyskany wskaźnik
```

3.4. Kompilowanie wyrażeń.

W kolejnych puntach tego rozdziału omówię sposob kompilacji wyrażeń poszczególnych typów.

3.4.1. Typy liczbowe.

W Adzie występuja trzy typy liczbowe: całkowite, rzeczywiste stałopozycyjne i rzeczywiste zmiennopozycyjne. W ramach niniejszej pracy zostały zaimplementowane tylko typy całkowitoliczbowe. W wyrażeniach arytmetycznych mogą występować funkcje zdefiniowane przez użytkownika oraz predefiniowane operatory.

Operatory, których argumenty mogą być liczbami całkowitymi, należą do pieciu grup. Grupy te (w kolejnosci rosnacego priorytetu) to: operatory relacyjne, addytywne binarne, addytywne unarne, multiplikatywne i operatory najwyższego priorytetu.

Operatory relacyjne: =, /=, <, <=, >, >= zostały zaimplementowane za pomocą instrukcji A-kodu: EQU, NEQ, LES, LEQ, GRE, GEQ. Przykładowo wyrażenie: (A >= 1) zostanie przetłumaczone na:

```
LODI   0   -8     -- załaduj wartość A
LDCI   1
GEQI
```

Litera 'I' na końcu mnemoniki instrukcji A-kodu oznacza, że argumenty tej instrukcji są typu całkowitego.

Binarne operatory addytywne: + i - zostały zaimplementowane za pomocą instrukcji ADD i SUB. Przykładowo wyrażenie: (A + 11) jest kompilowane na:

```
LODI   0   -8     -- załaduj wartość A
LDCI   11
ADDI
```

Jeśli typ wyrażenia nie jest standardowym typem INTEGER dodatkowo wygenerowany zostanie kod dla sprawdzenia, czy wyrażenie znajduje się w zadeklarowanym zakresie.
Np. przy założeniu deklaracji:

```
type SMALL_INT is range -200..200;
A: SMALL_INT;
```

wyrażenie: (A - 19) zostanie przetłumaczone na:

```
LODI   0   -4     -- załaduj wartość A
LDCI   19
SUBI
LDA    0    0     --  pobierz wskaźnik do opisu typu
                     SMALL_INT
CSP    CVB        -- i sprawdź zakres
```

Unarny operator + jest ignorowany. Operator - kompilowany jest z wykorzystaniem instrukcji NEG.
Np. wyrażenie (-A) jest tłumaczone na:

```
LODI   0   -4     -- załaduj wartość A
NEGI
```

Operatory multiplikatywne *, /, mod i rem są zaimplementowane za pomocą instrukcji MUL, DIV, MOD i REM.
Np. wyrażenie: ((5 * A) mod 7) zostanie przetłumaczone na:

```
LDCI   5
LODI   0   -4     -- załaduj wartość A
MULI
```

```
LDCI   7
MODI
```

Operatory najwyższego priorytetu ** i abs są tłumaczone z wykorzystaniem instrukcji A-kodu EXP i ABS.
Np. wyrażenie: (A ** 3) jest kompilowane na:

```
LODI   0   -4      -- załaduj wartość A
LDCI   3
EXPI
```

Warto zwrócić uwagę na to, że w oryginalnym P-kodzie niektóre operacje arytmetyczne były zaimplementowane za pomocą różnych instrukcji dla typu całkowitego i rzeczywistego. Np. dodawanie całkowitoliczbowe było realizowane za pomocą instrukcji ADI, a rzeczywiste - ADR. W niniejszej implementacji, zgodnie z sugestią zawartą w pracy [3], każdy operator reprezentowany jest za pomocą jednej instrukcji. Jej typ odróżniany jest za pomocą ostatniej litery mnemoniki instrukcji. Podejście to wydaje się być bardziej zgodne z ogólną koncepcją P-kodu, w którym większość instrukcji jest realizowana właśnie w ten sposób.

3.4.2. Typ logiczny.

Dla predefiniwanego typu BOOLEAN języka Ada zdefiniowanych jest aż dwanaście operatorów.

Operatory relacyjne kompilowane są tak samo jak dla wyrażeń całkowitoliczbowych. Jedyna różnicą jest stosowanie przyrostka 'B' zamiast 'I'.

Operator not tłumaczony jest za pomocą instrukcji NOT. Np. wyrażenie: (not B) kompilowane jest na:

```
LODB   0   -4      -- załaduj wartość B
NOT
```

Operatory logiczne and, or, xor są zaimplementowane za pomocą instrukcji AND, ORR, XOR.
Np. wyrażenie: (B and TRUE) tłumaczone jest na:

```
LODB   0   -4      -- załaduj wartość B
LDCB   TRUE
AND
```

Wyrażenia z operatorami and then i or else tłumaczone są za pomocą instrukcji skoku.
Np. wyrażenie (B1 and then B2) tłumaczone jest na:

```
    LODB   0   -4      -- załaduj wartość B1
    FJP    L    3
    LODB   0   -5      -- załaduj wartość B2
    UJP    L    4
L   3
    LDCB   FALSE
L   4
```

3.4.3. Typ znakowy.

W wyrażeniach typu CHARACTER mogą występować jedynie stałe, identyfikatory i funkcje zdefiniowane przez użytkownika. Kompilacja dokonywana jest za pomocą takich samych instrukcji jak dla typów całkowitych i logicznych. Jedyną różnicą jest używanie przyrostka 'C' oraz sposób zapisu stałych.

Np. stała 'A' zostaje przetłumaczona na instrukcję:

```
    LDCC    'A'
```

3.4.4. Typy wyliczeniowe.

Obiekty innych typów wyliczeniowych reprezentowane są jak obiekty całkowite. Kolejne zadeklarowane wartości typu przyjmują kolejne wartości całkowite, poczynając od zera.

Przykładowo przy założeniu deklaracji typu:

```
    type ENUM is ('A', EL);
```

wyrażenie: EL zostałoby przetłumaczone na:

```
    LDCI    1       -- EL
```

3.4.5. Typy wskazywane.

Obiekty typów wskazywanych traktowane są jak adresy A-kodu.

Stała null została zaimplementowana jako adres zerowy. Jest więc tłumaczona za pomocą instrukcji:

```
    LDCA    0       -- null
```

Operator new jest tłumaczony z wykorzystaniem instrukcji ALO. Np. wyrażenie: new INTEGER tłumaczone jest na:

```
    LDCI    2       --   rozmiar przydzielanego bloku pamięci
    ALO     0       --   żądanie przydziału
```

Pobranie obiektu wskazywanego przez zmienną typu access odbywa się za pomocą instrukcji IND (jeżeli obiekt jest typu skalarnego) lub GET (w przeciwnym wypadku).

Np. przy założeniu deklaracji:

```
    type A is access INTEGER;
    V: A;
```

wyrażenie: V.all zostaje skompilowane na:

```
LODA    0    0
INDI    0
```

3.4.6. Pozostałe typy.

Wyrażenia pozostałych typów (o ile nie są wywołaniami funkcji) tłumaczone są z wykorzystaniem instrukcji GET.

Np. przy założeniu deklaracji:

```
type A is array (1..10) of INTEGER;
V: A;
```

wyrażenie: V zostanie skompilowane na:

```
LDA     0    0
GET     8
```

3.4.7. Elementy tablic.

Dla obliczenia wyrażenia, które jest elementem tablicy, wyznaczany jest najpierw adres tego elementu. Następnie obiekt spod tego adresu jest pobierany, w sposób zależny od jego typu (np. za pomocą instrukcji IND).

Do wyliczenia adresu wykorzystywane są procedury standardowe AR1 i AR2.

Np. przy założeniu deklaracji:

```
type A is array (1..10, 2..20) of INTEGER;
V: A;
```

wyrażenie: V(5, 15) tłumaczone jest na:

```
LDA     0    -16     --   adres zmiennej V
DPLA
INDA    0            --   wskaźnik do wartości V
SWPA
INDA    -4           --   wskaźnik do opisu
DECA    2            --   ustaw wskaźnik na opis 1. wymiaru
LDCI    5
CSP     AR1          --   oblicz przesunięcie w 1. wymiarze
DECA    6            --   ustaw wskaźnik na opis 2. wymiaru
LDCI    15
CSP     AR2          --   oblicz przesunięcie w 2. wymiarze
ADDI                 --   dodaj przesunięcia
IXA     1            --   dodaj przesunięcie do adresu bazowego
INDI    0            --   i pobierz wartość
```

3.5. Kompilowanie instrukcji.

3.5.1. Podstawienie.

3.5.1.1. Podstawienie pod bezpośrednią zmienną prostą, np.:

```
a := 0;
```

Jeśli a jest zmienną lokalną:

```
   LDCI    0          -- załaduj zero
   STRI    0    5     -- prześlij do a
```

Jeśli a jest zmienną globalną:

```
   LDCI    0
   SROI    0    8
```

3.5.1.2. Tablice.

Przypisanie obiektu tablicowego tłumaczone jest z wykorzystaniem procedury CYA. Np. instrukcja:

```
   -- type T is array (5..15) of INTEGER;
   -- A, B : T;
   A := B;
```

jest tłumaczona na:

```
   LAO     0   -8      -- załaduj adres A
   LAO     0  -16      -- załaduj adres B
   LDCI    1           -- ilość wymiarów
   CSP     CYA         -- i wywołaj procedurę kopiujacą tablicę
```

Jeśli po prawej stronie instrukcji przypisania znajduje się wyrażenie, wykorzystywana jest procedura PUA. Np. instrukcja:

```
   -- type T is array (5..15) of INTEGER;
   -- A, B : T;
   -- function "+"(L, R: T) return T is ...
   A := B + A;
```

jest tłumaczona na:

```
   LDA     0   -8        -- załaduj adres A
   MST     8    0
   LDA     0  -16        -- załaduj adres B
   GET     8
   LDA     0   -8        -- załaduj adres A
   GET     8
   CUP    18    L   3    -- wywołaj funkcję "+"
   LDCI    1             -- załaduj ilość wymiarów
   CSP     PUA           -- i wywołaj procedurę kopiujacą tablicę
```

W obu powyższych przypadkach możnaby wykorzystać ta samą procedurę PUA. Procedura CYA została wprowadzona po to, by w tym szczególnym wypadku umożliwić wygenerowanie bardziej optymalnego kodu.

3.5.2. Instrukcja warunkowa.

Instrukcja:

```
   if warunek 1 then
    ciag instrukcji 1
   elsif warunek 2 then
    ciag instrukcji 2
   else
    ciag instrukcji 3
   end if;
```

zostaje przetłumaczona na:

```
    [ kod dla warunku 1 ]
    FJP    L    6      -- skok  do  etykiety  6  gdy warunek 1 nie
                          jest spełniony
    [ kod dla ciagu instrukcji 1 ]
    UJP    L    8
L   6
    [ kod dla warunku 2 ]
    FJP    L    7      -- skok do etykiety  7  gdy  warunek  2 nie
                          jest spełniony
    [ kod dla ciagu instrukcji 2 ]
    UJP    L    8
L   7
    [ kod dla ciagu instrukcji 3 ]
L   8
```

Dla instrukcji warukowej bez części else:

```
    if warunek then
     ciag instrukcji
    end if;
```

jest generowany nastepujący kod:

```
    [ kod dla warunku ]
    FJP     L    6    -- skok do etykiety 6 jeśli warunek nie był
spełniony
    [ kod dla ciagu instrukcji ]
L   6
```

3.5.3. Instrukcja wyboru.

Instrukcja:

```
    case wyrażenie of
      when 2 | 4  => ciag instrukcji 1
      when 3 | 6  => ciag instrukcji 2
      when others => ciag instrukcji 3
    end case
```

zostaje przetłumaczona na:

```
    [ kod dla wyrażenia ]
    UJP    L    8
L   10
    [ kod dla ciagu instrukcji 1 ]
    UJP    L    9       -- skok do instrukcji końcowej
L   11
    [ kod dla ciagu instrukcji 2 ]
    UJP    L    9
L   12
    [ kod dla ciagu instrukcji 3 ]
    UJP    L    9
L   8
    DECI   2
    XJP    5            --   jeśli  wartosc  >=  5  lub  ujemna to
                           wykorzystaj   ostatnia   etykietę  na
                           liscie.
    UJP    L    10      -- wyrażenie = 2
    UJP    L    11      -- wyrażenie = 3
    UJP    L    10      -- wyrażenie = 4
    UJP    L    12      -- wyrażenie = 5
    UJP    L    11      -- wyrażenie = 6
    UJP    L    12      -- wyrażenie < 2 lub wyrażenie > 6
L   9
```

3.5.4. Instrukcje pętli.

3.5.4.1. Instrukcja loop.

Instrukcja:

```
    loop
      ciąg instrukcji
    end loop;
```

zostaje przetłumaczona na:

```
L   8
    [ kod dla ciągu instrukcji ]
    UJP    L    8
```

3.5.4.2. Instrukcja while.

Instrukcja:

```
    while warunek loop
      ciąg instrukcji
    end loop;
```

zostaje przetłumaczona na:

```
L   8
    [ kod dla warunku ]
    FJP    L    9
    [ kod dla ciągu instrukcji ]
    UJP    L    8
L   9
```

3.5.4.3. Instrukcja for.

3.5.4.3.1. Instrukcja:

```
    for j in m..n loop
      ciąg instrukcji
    end loop;
```

zostaje przetłumaczona na:

```
    [ kod dla m ]
    STRI   0    j
    [ kod dla n ]
    STRI   temp
L   8
    LODI   0    j
    LODI   0    temp
    LEQI           -- czy j <= temp
    FJP    L     9
    [ kod dla ciągu instrukcji ]
    LODI   0    j
    INCI   1      -- inkrementacja j
    STRI   0    j
    UJP    L    8
L   9
```

3.5.4.3.2. Instrukcja:

```
    for j in reverse m..n loop
     ciąg instrukcji
    end loop;
```

zostaje przetłumaczona na:

```
    [ kod dla m ]     -- zmienna temp
```

```
   STRI   0    temp
   [ kod dla n ]     -- zmienna j
L  8
   STRI   0    j
   LODI   0    j
   LODI   0    temp
   GEQI              -- czy j >= temp
   FJP    L    9
   [ kod dla ciągu instrukcji ]
   LODI   0    j
   DECI   1         -- dekrementacja j
   UJP    L    8
L  9
```

Jeśli zmienna robocza pętli jest typu znakowego lub logicznego, to w powyższych sekwencjach przyrostki 'I' instrukcji A-kodu są zastąpione odpowiednio przez 'C' lub 'B'.

Pamięć dla zmiennych 'j' i 'temp' w powyższych schematach jest rezerwowana w obszarze zmiennych lokalnych bloku bezpośrednio otaczającym instrukcję pętli **for**. Pamięć ta jest odzyskiwana, tzn. dla danych organizacyjnych dwóch kolejnych instrukcji **for** przydzielany jest ten sam obszar pamięci.

3.5.5. Instrukcja bloku.

Instrukcja bloku traktowana jest jak bezparametrowa procedura. Np., instrukcja:

```
   declare
     deklaracje
   begin
     ciąg instrukcji
   end;
```

zostaje przetłumaczona na:

```
   MST    0    0
   CUP    2    L    4
   UJP    L    5
L  4
   ENT    1    L    6
   ENT    2    L    7
   [ kod dla deklaracji ]
   EXH    L    8
   [ kod dla ciągu instrukcji ]
L  9
   RET    2
L  8
   EEX
L  6   =  [ rozmiar zmiennych lokalnych ]
L  7   =  [ maksymalny obszar zajmowany przez ramkę bloku ]
L  5
```

3.5.6. Instrukcja wyjścia z pętli.

3.5.6.1. Jeżeli jednocześnie z wyjściem z pętli nie następuje wyjście z bloku, generowana jest tylko instrukcja skoku bezwarunkowego. Np. instrukcja:

```
loop
  ciąg instrukcji 1
  exit;
  ciąg instrukcji 2
end loop;
```

zostaje przetłumaczona na:

```
L   8
    [ kod dla ciągu instrukcji 1 ]
    UJP    L    9
    [ kod dla ciągu instrukcji 2 ]
    UJP    L    8
L   9
```

3.5.6.2. Jeżeli jednocześnie z wyjściem z pętli następuje wyjście z bloku, to przed instrukcją skoku bezwarunkowego generowana jest instrukcja opuszczenia bloku LVB (porównaj przykład dla skoku na zewnątrz bloku - punkt 3.5.8.2).

3.5.7. Instrukcja powrotu z podprogramu.

Instrukcja ta jest tłumaczona jako skok bezwarukowy (UJP) do etykiety znajdującej się przed częścią podprogramu kopiująca na zewnątrz parametry skalarne trybu **in** i **in out** znajdująca się przed instrukcją RET. Ewentualny parametr instrukcji **return** jest oczywiście najpierw wyliczany i umieszczany w obszarze wartości zwracanej przez funkcję. Jeżeli jednocześnie następuje wyjście z bloku, przed instrukcją skoku bezwarunkowego generowana jest instrukcja opuszczenia bloku (LVB).

3.5.8. Instrukcja skoku.

3.5.8.1. Jeżeli skoku nie powoduje opuszczenia bloku, generowana jest tylko instrukcja skoku bezwarunkowego.

```
UJP    L    7
```

3.5.8.2. W przypadku skoku na zewnątrz bieżącego bloku, przed instrukcją skoku jest generowana instrukcja wyjścia z bloku LVB, np. procedura:

```
procedure Test11 is
begin
  ciąg instrukcji 1
  begin
    ciąg instrukcji 2
    goto LBL;
```

```
        ciąg instrukcji 3
      end;
      <<LBL>>
      ciąg instrukcji 4
    end Test11;
zostaje przetłumaczona na:
L   1
    ENT    1    L    2
    ENT    2    L    3
    [ kod dla ciągu instrukcji 1 ]
    MST    2    0
    CUP    0    L    4
    UJP    L    5
L   4
    ENT    1    L    6
    ENT    2    L    7
    [ kod dla ciągu instrukcji 2 ]
    LVB    L    8
    UJP    L    9
    [ kod dla ciągu instrukcji 3 ]
    RET    2
L   6   =  0
L   7   =  0
L   5
L   8   =  1                    -- należy opuscic jeden blok
L   9
    [ kod dla ciągu instrukcji 4 ]
    RET    2
L   2   =  0
L   3   =  0
```

3.6. Kompilowanie procedur i funkcji.

3.6.1. Kompilowanie wywołań procedur i funkcji.

3.6.1.1. Wywołanie procedury (lub funkcji) jest realizowane za pomoca nastepujacego ciagu instrukcji:

```
    MST    2    0
    [ instrukcje umieszczajace parametry na stosie ]
    CUP    0    L    3
```

Pierwszy parametr rozkazu MST określa rozmiar obiektu zwracanego przez funkcje, a drugi - zanurzenie danej procedury. Dokładnie mowiac drugi parametr jest zdefiniowany jako "jeden plus poziom wywołujacej procedury minus poziom wywoływanej procedury". Np. w następującej procedurze liczby w komentarzach oznaczaja wartosci drugiego parametru instrukcji MST, które sa generowane dla zrealizowania wywołania procedury:

```
    procedure A is
      procedure B is
      begin
        null;
      end B;
      procedure C is
        procedure D is
```

```
      begin
        A;    -- 3
        B;    -- 2
        C;    -- 2
        D;    -- 1
      end D;
    begin      -- C
      B;    -- 1
      C;    -- 1
      D;    -- 0
    end C;
  begin      -- A
    B;    -- 0
    C;    -- 0
  end A;
```

3.6.1.2. Wywołanie procedury z parametrami skalarnymi w trybie in lub in out wymaga umieszczenia na stosie także adresów parametrów. Raport języka Ada definiuje bowiem, że takie parametry maja być przekazywane przez kopiowanie (LRM 6.2).

Np. przy założeniu następujących deklaracji:

```
   -- Z1, Z2, Z3: INTEGER;
   -- procedure P( A: INTEGER;
   --              B: in out INTEGER;
   --              C: out INTEGER ) is ...
```

wywołanie:

```
   P( Z1, Z2, Z3);
```

zostaje przetłumaczone na:

```
   MST    2    0
   LODI   1    10
   LDA    1    12
   DPLA
   INDI   0
   LDA    1    14
   DPLA
   INDI   0
   CUP    14   L   7
```

Jako alternatywne rozwiązanie rozpatrywałem także, możliwość umieszczania na stosie tylko adresów parametrów aktualnych. Kopiowanie odbywałoby się wtedy wewnątrz wywoływanego podprogramu. Rozwiązanie takie powodowałoby wygenerowanie krótszego kodu (sekwencja kopiująca wystąpiłaby tylko raz, a nie przed każdym wywołaniem). Przyjęte rozwiązanie jest za to szybsze.

3.6.2. Kompilowanie deklaracji procedur.

Przydział pamięci dla parametrów i zmiennych lokalnych jest opisany w punkcie 3.2 mojej pracy (tzn. parametry mają wartości przesunięć dodatnie, a zmienne lokalne - ujemne).

W przypadku parametrów skalarnych w trybach in i in out, umieszczone na stosie przed wywołaniem adresy parametrów umożliwiają przed zokończeniem procedury skopiowanie lokalnych wartości na zewnątrz.

Np. deklaracja procedury:

```
procedure P( A: in INTEGER;
             B: in out INTEGER;
             C: out INTEGER) is
begin
  C := A + B;
  B := 1;
end P;
```

zoataje przetłumaczona na:

```
L   8
    ENT    1    L   9
    ENT    2    L   10
    EXH    L    11
    LODI   0    24      -- załaduj A
    LODI   0    18      -- załaduj B
    ADDI                -- i dodaj
    STRI   0    12      -- wynik umieść w C
    LDCI   1            -- załaduj stałą 1
    STRI   0    18      -- i umieść ją w B
L   12                  -- tu rozpoczyna się kopiowanie parametrów
    LODA   0    22      -- załaduj adres parametru aktualnego B
    LODI   0    18      -- załaduj wartość B
    STOI                -- i prześlij ją
    LODA   0    16      -- to samo z parametrem C
    LODI   0    12
    STOI
    RET   16            --   parametr   określa   rozmiar   obszaru
                             parametrów procedury
L   11
    EEX
L   9     =  14
L   10    =  18
```

W tym przykładzie przydział pamięci dla zmiennych lokalnych i parametrów jest następujący:

24: 23:	Parametr aktualny A
22: 21: 20: 19:	Adres parametru aktualnego B
18: 17:	Parametr aktualny B
16: 15: 14: 13:	Adres parametru aktualnego C
12: 11:	Parametr aktualny C

3.6.3. Kompilowanie deklaracji funkcji.

W przypadku kompilacji funkcji należy dodatkowo uwzględnić wartość zwracana przez funkcję. Wartość ta jest oddzielona od parametrów liczbą, określającą względne zanurzenie wywoływanego podprogramu. Liczba ta służy następnie do utworzenia łącza statycznego. Ponieważ (zgodnie z definicją P-kodu) jest ona podawana za pomocą instrukcji MST, a łącze statyczne korzystniej jest wyliczyć wewnątrz wywoływanego podprogramu, wartość ta musi być przechowywana.

Przykładowo przy założeniu wcześniejszej deklaracji typu:

```
    -- type A is array (1..20);
deklaracja:
    function "+"(LEFT, RIGHT: A) return A is
      R: A;
    begin
      [ ciąg instrukcji ]
      return R;
    end "+";
```

zostaje przetłumaczona na:

```
L   11
    ENT    1    L   13
    ENT    2    L   14
-- inicjalizuj tablicę zwracana jako wartosc:
    LAO    2    0
    DPLA
    STRA   0    32                -- ^opis
    INDI   0
    ALO    -1                     -- przydziel pamiec
    STRA   0    36                -- ^wartosc
-- inicjalizuj tablice R:
    LAO    2    0
    DPLA
```

```
    STRA   0    -4                  -- ^opis
    INDI   0
    ALO    0                        -- przydziel pamięć
    STRA   0    0                   -- ^wartość
    EXH    L    15                  -- begin
    [ kod dla ciagu instrukcji ]
    LDA    0    36                  -- wartość funkcji
    LDA    0    0                   -- R
    GET    8                        -- umieść R na stosie
    LDCI   2                        -- liczba wymiarów
    CSP    PUA                      -- kopiuj tablicę
L   12
    RET    18
L   15
    EEX
L   13     =    8
L   14     =    22
```

Przydział pamięci dla parametrów, zmiennych lokalnych i wartości funkcji jest w tym przykładzie następujący:

36:	^wartość	wartość
32:	^opis	funkcji
28:	wzgl. zagłębienie funkcji	
26:	^wartość	LEFT
22:	^opis	
18:	^wartość	RIGHT
14:	^opis	
	adres powrotu itd.	
0:	^wartość	R
-4:	^opis	

3.7. Rozłączna kompilacja.

Punkt startowy jednostki kompilacji oznaczony jest etykietą o numerze 1. Gdy jednostką kompilacji jest procedura, punkt startowy jest interpretowany jako początek tej procedury, w przeciwnym wypadku jest on interpretowany jako początek procedury inicjalizującej daną jednostkę kompilacji.

Treść każdej jednostki kompilacji jest kompilowana jak procedura użytkownika A-kodu (bezparametrowa w przypadku pakietów).

Klauzula with kompilowana jest jako ciąg opisów używanych jednostek bibliotecznych. Dla każdej jednostki wymienionej w klauzuli with generowany jest tylko jeden opis. Pierwszą instrukcją opisu jest RFP. Instrukcja ta wiąże z wymienioną jednostką numer wykorzystywany następnie w instrukcjach dostępu do zmiennych globalnych - zadeklarowanych na poziomie głównym pakietu. Następnie jeśli jednostką opisywaną jest procedura, występuje pojedyncza instrukcja RFL utożsamiająca etykietę z tą procedurą. Jeśli natomiast opisywaną jednostką jest pakiet, to po instrukcji RFP następuje ciąg instrukcji EXL i RFL utożsamiających etykiety z wyjątkami i procedurami zadeklarowanymi w części widocznej pakietu. Występują one zawsze w tej samej kolejności - w kolejności deklaracji w części widocznej pakietu.

3.7.1. Pakiet jako jednostka kompilacji.

Pierwsza instrukcja skompilowanego pakietu jest dyrektywa:

PKG <nazwa pakietu>

Po niej następują opisy pakietów występujących w klauzuli with (o ile była obecna), oraz treść procedury inicjalizującej. Deklaracje są kompilowane zgodnie z ogólnymi zasadami. Np. pakiet:

```
   package Test2 is
     I : INTEGER := 3;
     E1, E2 : exception;
     procedure P( L : CHARACTER );
     procedure P( L : INTEGER );
   end Test2;
```

zostaje przetłumaczony na:

```
   PKG    TEST2
   RFP    1    STANDARD
   [ definicje etykiet pakietu STANDARD ]
L  1
   ENT    1    L    2
   ENT    2    L    3
   LDCI   3
   SROI   0    0          -- inicjalizuj zmienna I
   EXH    L    4
   RET    2
L  4
   EEX
L  2  =   2
L  3  =   6
   Q
```

3.7.2. Treść pakietu jako jednostka kompilacji.

Pierwszą instrukcją skompilowanej treści pakietu jest dyrektywa:

PKB <nazwa pakietu>

Po niej następują opisy pakietów występujących w klauzuli with (o ile była obecna), treść procedury inicjalizującej oraz kod treści pakietu. Np.:

```
package body Test2 is
  J : INTEGER := 0;
  procedure P( L : CHARACTER ) is
  begin
    null;
  end P;
  procedure P( L : INTEGER ) is
  begin
   J := L;
  end P;
end Test2;
```

Test2:

0:	I
-1:	
-2:	J
-3:	

zostaje przetłumaczony na:

```
    PKB     TEST2
    EXL     L     15     E1   -- Definicje etykiet odpowiadających
    EXL     L     16     E2   -- wyjątkom zdefiniowanym w pakiecie.
    RFL     L     2           --      Definicje etykiet procedur w
                                      takiej kolejności w jakiej
                                      wystąpiły deklaracje w
                                      pakiecie.
    RFL     L     3
    RFP     1     STANDARD
    [definicje etykiet dla pakietu STANDARD]
L   1
    ENT     1     L     4
    ENT     2     L     5
    LDCI    0
    SROI    0     -2          -- inicjalizuj zmienna J
    UJP     L     6           -- omiń treść procedury
L   2                         -- procedure P
    ENT     1     L     7
    ENT     2     L     8
    EXH     L     9
L   10
    RET     2
L   9
    EEX
L   7     =   0
L   8     =   0
L   3                         -- procedure P
    ENT     1     L     11
    ENT     2     L     12
    EXH     L     13
    LODI    0     12          -- pobierz wartość parametru
    SROI    0    -2           -- i prześlij do zmiennej J
L   14
    RET     2
L   13
    EEX
L   11    =   2
```

```
L   12   =   4
L   6
    EXH     L     14
    RET     2
L   14
    EEX
L   4    =   4
L   5    =   6
    Q
```

3.7.3. Procedura jako jednostka kompilacji.

Pierwszą instrukcją skompilowanej procedury jest dyrektywa:

PRO <nazwa procedury>

Po niej następują opisy pakietów występujących w klauzuli with (o ile była obecna) oraz kod treści procedury.

Np. jednostka kompilacji:

```
with Test2; use Test2;
procedure Test3 is
  procedure T( J : out INTEGER ) is
  begin
    J := 7;
  end T;
begin
  T( I );
end Test3;
```

Zostaje przetłumaczona na:

```
    PRO     TEST3
    RFP     1     STANDARD
    [definicje etykiet dla pakietu STANDARD]
    RFP     2     TEST2
    EXL     L     13     E1
    EXL     L     14     E2
    RFL     L     2
    RFL     L     3
L   1
    ENT     1     L     9
    ENT     2     L     10
    UJP     L     13          -- omin tresc procedury
L   4                         -- procedure T
    ENT     1     L     5
    ENT     2     L     6
    EXH     L     7
    LDCI    7
    STRI    0     12          -- J
L   8                         -- skopiuj parametr J na zewnatrz
    LODA    0     16
    LODI    0     12
    STOI
    RET     6
L   7
    EEX
L   5    =   6
L   6    =   12
L   13
    EXH     L     11          -- begin Test3
```

```
    MST     0     0
    LAO     1     0
    DPLA
    INDI    0
    CUP     6     L     4     -- T
L   12
    RET     2
L   11
    EEX
L   9   =   0
L   10  =   0
    Q
```

3.8. Kompilowanie zadań.

3.8.1. Deklaracje.

3.8.1.1. Deklaracja typu zadaniowego.

Zadeklarowanie typu zadaniowego powoduje utworzenie w obszarze zmiennych lokalnych bieżącego bloku struktury opisującej typ zadaniowy. Pierwszym elementem (typu INTEGER) jest liczba wejsc zadania. Po niej następują pary liczb opisujące każde wejście. Pary te opisuja zakres zmian indeksu rodziny wejść. Wejscia pojedyncze traktowane są jak rodzina wejść o zakresie 1..1. Np. deklaracja:

```
task type T is
  entry E1(3..17) (I : in  INTEGER);
  entry E2        (I : out INTEGER);
end T;
```

zostaje przetłumaczona na:

```
    LDCI    3             -- dolna   granica   zakresu   pierwszego
                             wejscia
    STRI    0    -2
    LDCI    17
    STRI    0    -4
    LDCI    1
    STRI    0    -6
    LDCI    1
    STRI    0    -8
    LDCI    2             -- liczba wejsc
    STRI    0     0
```

3.8.1.2. Deklaracje zadań.

Każde zadeklarowane zadanie, czy to pojedyncze, czy to obiekt typu zadaniowego, powoduje wygenerowanie instrukcji ACT i zapamiętanie wskaźnika do opisu zadania. Zadania nie sa rzeczywiście aktywizowane, jeśli dla danego bloku nie została jeszcze wykonana instrukcja EXH. Tak wiec rzeczywista aktywizacja wszystkich zadań zostaje wykonana przez procedury RUNTIME dopiero jako jeden z efektów wykonania instrukcji EXH. Takie zachowanie

wynika z konieczności inicjalizacji zadań zgodnie ze standardem języka Ada (LRM 9.3/2).

Np. deklaracja:

```
   -- task type T is
   --   entry E1(3..17) (I : in  INTEGER);
   --   entry E2        (I : out INTEGER);
   -- end T;
   T1: T;
```

zostaje przetłumaczona na:

```
   LDA    0    0          -- wskaźnik do opisu typu zadaniowego
   ACT    0    L    3
   STRA   0    10
```

3.8.1.3. Aktywizacja zadań wskazywanych.

Do aktywizacji zadań wskazywanych wykorzystywana jest instrukcja ACA, która powoduje natychmiastową aktywizacja zadania.

Np. instrukcja:

```
   -- task type T is
   --   entry E1(3..17) (I : in  INTEGER);
   --   entry E2        (I : out INTEGER);
   -- end T;
   -- type A is access T;
   -- T1: A;
   T1 := new T;
```

zostanie przetłumaczona na:

```
   LDA    0    0          -- wskaźnik do opisu typu zadaniowego
   LDCI   0               -- zagłębienie typu A
   ACA    0    L    3
   STRA   0    10
```

3.8.1.4. Deklaracja treści zadania.

Treść zadania tłumaczona jest jak procedura bezparametrowa. Jedyna różnica jest generowanie instrukcji ETK zamiast RET.

Ogolnie - tresc:

```
   task body T is
     deklaracje
   begin
     ciag instrukcji
   exception
     obsługa wyjatków
   end T;
```

Zostaje przetłumaczona na:

```
L  3
   ENT    1    L    4
   ENT    2    L    5
   [ kod dla deklaracji ]
   EXH    L    6
   [ kod dla ciagu instrukcji ]
L  7
   ETK
L  6
```

```
[ kod dla obsługi wyjątków ]
EEX
```

3.8.2. Wejścia (instrukcje accept, select).

3.8.2.1. Prosta instrukcja accept.

Aby umożliwić przekazywanie parametrów wejść wykorzystując mechanizm przekazywania parametrów stosowany dla procedur i funkcji, wywołanie wejścia powoduje otwarcie nowej ramki (zawsze bez zmiennych lokalnych i obsługi wyjątków). Instrukcja accept kompilowana jest więc jak bezparametrowa procedura. Np. instrukcja:

```
-- task T is
--   entry E1(3..17) (I : in  INTEGER);
--   entry E2        (I : out INTEGER);
-- end T;
-- task body T is
-- begin
     accept E1(7) (I : in INTEGER) do
     begin
       ciag instrukcji
     end E1;
-- end T;
```

zostaje przetłumaczona na:

```
    LDCI   1                -- numer wejścia
    LDCI   7                -- numer w ramach rodziny
    ACC
    ENT    1    L    4
    ENT    2    L    5
    EXH    L    6
    UJP    L    7
L   6
    EEX
L   7
    [kod dla ciagu instrukcji]
    EAC
```

3.8.2.2. Instrukcja select z opcją else.

Np. instrukcja:

```
-- task T is
--   entry E1;
--   entry E2;
-- end T;
-- task body T is
-- begin
     select
       accept E1;
       ciag instrukcji 1
     or
       accept E2;
       ciag instrukcji 2
     else
       ciag instrukcji 3
     end select;
```

```
   -- end T;
zostaje przetłumaczona na:
   LDCI   1                -- numer wejścia E1
   LDCI   1                -- numer w ramach rodziny
   UJP    L    2           -- omiń treść opcji
L  3
   ENT    1    L    4      -- accept
   ENT    2    L    5
   EXH    L    6
   UJP    L    7
L  6
   EEX
L  7
   [ kod dla ciągu instrukcji 1 ]
   EAC
   UJP    L    8
L  2
   LDCI   2                -- numer wejścia E2
   LDCI   1                -- numer w ramach rodziny
   UJP    L    9           -- omiń treść opcji
L  10
   ENT    1    L    11     -- accept
   ENT    2    L    12
   EXH    L    13
   UJP    L    14
L  13
   EEX
L  14
   [ kod dla ciągu instrukcji 2 ]
   EAC
   UJP    L    8
L  15
   [ kod dla ciagu instrukcji 3 ]
L  9
   SLE    2
   XJP    2
   UJP    L    3           -- opcja 1. (accept E1)
   UJP    L    10          -- opcja 2. (accept E2)
   UJP    L    15          -- opcja 3. (else)
L  8
```

3.8.2.3. Instrukcja select bez opcji else, delay, terminate.

Instrukcja taka jest tłumaczona tak jak instrukcja select z opcja else. Jedynymi roznicami sa: brak ciagu instrukcji odpowiadajacego opcji else oraz uzycie instrukcji SLA zamiast SLE.

3.8.2.4. Instrukcja select z opcja terminate.

Instrukcja taka jest tłumaczona tak jak instrukcja select z opcja else. Jedynymi roznicami sa: brak ciagu instrukcji odpowiadajacego opcji else oraz uzycie instrukcji SLT zamiast SLE.

3.8.2.5. Instrukcja select z opcją delay.

Instrukcja taka jest tłumaczona tak jak instrukcja select z ɔcją else. Jedynymi różnicami są: umieszczenie na szczycie stosu ɔdatkowo liczby odpowiadającej maksymalnemu parametrowi ıstrukcji delay, zastąpienie ciągu instrukcji odpowiadającego ɔcji else ciągiem instrukcji odpowiadającym opcji delay oraz ;ycie instrukcji SLD zamiast SLE.

ɔ. instrukcja:

```
-- task T is
--   entry E1;
--   entry E2;
-- end T;
-- task body T is
-- begin
     select
       accept E1;
       ciag instrukcji 1
     or
       accept E2;
       ciag instrukcji 2
     or
       delay 500;
       ciag instrukcji 3
     end select;
-- end T;
```

ıstaje przetłumaczona na:

```
LDCI   1                  -- numer wejścia E1
LDCI   1                  -- numer w ramach rodziny
UJP    L    2             -- omiń treść 1. opcji
3
ENT    1    L    4        -- accept
ENT    2    L    5
EXH    L    6
UJP    L    7
6
EEX
7
[ kod dla ciagu instrukcji 1 ]
EAC
UJP    L    8
2
LDCI   2                  -- numer wejścia
LDCI   1                  -- numer w ramach rodziny
UJP    L    16            -- omiń treśc 2. opcji
10
ENT    1    L    11       -- accept
ENT    2    L    12
EXH    L    13
UJP    L    14
13
EEX
14
[kod dla ciagu instrukcji 2]
```

```
    EAC
    UJP     L     8
L   16
    LDCI    -2                -- opcja delay
    LDCI    500
    UJP     L     9
L   15
    [ kod dla ciagu instrukcji 3 ]
L   9
    SLD     2
    XJP     2
    UJP     L     3           -- opcja 1. (accept E1)
    UJP     L     10          -- opcja 2. (accept E2)
    UJP     L     15          -- opcja 3. (delay 500)
L   8
```

Uwaga:

Ponieważ w niniejszej implementacji nie występują liczby rzeczywiste parametr instrukcji **delay** jest parametrem całkowitym. Zgodnie z definicją języka parametrem powinno być jednak wyrażenie predefiniowanego typu DURATION (LRM 9.6).

3.8.3. Wywołanie wejścia.

Wywołanie wejścia zostaje przygotowane tak, jak wywołanie procedury (instrukcja MST z następującym po niej kodem powodującym umieszczenie na stosie parametrów). Następnie na stosie zostaje umieszczony wskaźnik do opisu zadania, numer wejścia i numer w ramach rodziny (a dla instrukcji ETD dodatkowo liczba odpowiadająca parametrowi instrukcji **delay**). Instrukcjami realizującymi wywołanie wejścia są: ETD, ETE i ETR.

3.8.3.1. Proste wywołanie wejścia.

Np. instrukcja:

```
    -- task T is
    --    entry E (I : INTEGER);
    -- end T;
    T.E(97);
```

zostaje przetłumaczona na:

```
    MST     0     0
    LDCI    97                -- parametr
    LODA    0     0           -- wskaznik do zadania
    LDCI    1                 -- numer wejscia
    LDCI    1                 -- numer w ramach rodziny
    ETR     2                 -- wywołaj wejscie
```

3.8.3.2. Instrukcja selektywnego wejścia z opcją else.

Np. instrukcja:

```
    -- task T is
```

```
   --   entry E;
   -- end T;
   select
     T.E;
     ciąg instrukcji 1
   else
     ciąg instrukcji 2
   end select;
```

Zostaje przetłumaczona na:

```
   MST    0    0
   LODA   0    0          -- wskaźnik do zadania
   LDCI   1               -- numer wejścia
   LDCI   1               -- numer w ramach rodziny
   ETE    0               -- parametr oznacza rozmiar obszaru
                             parametrów wejścia
   FJP    L    2
   [kod dla ciągu instrukcji 1]
   UJP    L    3
L  2
   [kod dla ciągu instrukcji 2]
L  3
```

3.8.3.3. Instrukcja selektywnego wejścia z opcją delay.

Np. instrukcja:

```
   -- task T is
   --   entry E;
   -- end T;
   select
     T.E;
     ciąg instrukcji 1
   or
     delay 1000;
     ciąg instrukcji 2
   end select;
```

Zostaje przetłumaczona na:

```
   MST    0    0
   LODA   0    0          -- wskaźnik do zadania
   LDCI   1               -- numer wejścia
   LDCI   1               -- numer w ramach rodziny
   LDCI   1000
   ETD    0
   FJP    L    2
   [ kod dla ciągu instrukcji 1 ]
   UJP    L    3
L  2
   [ kod dla ciągu instrukcji 2 ]
L  3
```

3.8.4. Instrukcja abort.

Instrukcja abort jest tłumaczona bezpośrednio jako ciąg instrukcji powodujący umieszczenie na stosie wskaźnika do opisu zadania i następującej po nim instrukcji ABO. Np. instrukcja:

```
   -- task T is
   --   entry E (I : in  INTEGER);
   -- end T;
```

```
  abort T;
staje przetłumaczona na:
  LODA   0    8          -- wskaźnik do opisu zadania
  ABO
```

3.8.5. Atrybuty zadań.

Wszystkie atrybuty zadań (CALLABLE, COUNT i TERMINATED) są implementowane jako wywołania procedur standardowych dpowiednio: CLB, CNT i TRM).

.9. Kompilowanie wyjątków.

3.9.1. Deklarowanie wyjątków.

W wyniku deklaracji wyjątku generowana jest instrukja EXL definiująca równoważnosc między zadeklarowanym wyjątkiem a etykieta. Od tego miejsca we wszystkich instrukcjach dotyczacych tego wyjatku wykorzystywana jest ta etykieta.

3.9.2. Zgłaszanie wyjątków.

Instrukcja **raise** języka Ada powoduje wygenerowanaie instrukcji RAI p-kodu. Argumentem jej jest etykieta przydzielona instrukcją EXL. Instrukcja **raise** bezparametrowa zostaje przetłumaczona na bezparametrową instrukcję RAI.

3.9.3. Obsługa wyjątków.

Po zakończeniu deklaracji danego bloku generowana jest instrukcja EXH, której argument wskazuje na segment obsługi wyjatków. Segment obsługi wyjatków kompilowany jest na ciag instrukcji EXC odpowiadajacych każdemu obsługiwanemu wyjątkowi oraz instrukcji EEX kończacej segment obsługi wyjątków.

3.9.4. Przykład. Procedura:

```
  procedure Test20 is
    E1, E2 : exception;
  begin
    ciag instrukcji 1
    raise E1;
    ciag instrukcji 2
  exception
    when E1 =>
      null;
    when E2 =>
      ciag instrukcji 3
      raise;
  end Test20;
```

zostaje przetłumaczona na:

```
1
ENT    1    L    2
ENT    2    L    3
EXL    L    4    E1
EXL    L    5    E2
EXH    L    6
[kod dla ciągu instrukcji 1]
RAI    L    4                    -- raise E1
[kod dla ciągu instrukcji 2]
7
RET    0
6
EXC    L    4     L    8
UJP    L    7
8
EXC    L    5     L    9
[kod dla ciągu instrukcji 3]
RAI
9
EEX
```

4. Opis implementacji.

4.1. Wykorzystywane struktury danych.

Strukturą danych, która dostarcza wszelkich danych o kompilowanej jednostce oraz służy do przechowywania wszystkich danych związanych z generacją kodu jest drzewo DIANY. Jest to reprezentacja kompilowanej jednostki, wyrażona w specjalnym języku pośrednim - w DIANIE. Opis tego języka znajduje się w rozdziale 2 mojej pracy.

Sposób wewnętrznej reprezentacji kompilowanej jednostki, z wykorzystaniem notacji DIANA, został zaprojektowany tak, by jak najlepiej wykorzystać możliwości kompilatora Turbo Pascal 5.5. Jej definicja, wykorzystywana w tym programie, znajduje się w module 'Diana'. Implementacja struktury DIANY wykonana jest z użyciem trzech typów: 'NODE_NAME', 'NODE' i 'TREE'.

Typ wyliczeniowy 'NODE_NAME' reprezentuje wszystkie możliwe rodzaje węzłów DIANY. Identyfikatory oznaczające wartości tego typu zawierają nazwę węzła poprzedzoną znakiem podkreślenia. Każdemu węzłowi odpowiada też definicja typu rekordowego. Nazwa typu składa się z nazwy węzła i przedrostka 'r_'. Pola poszczególnych rekordów odpowiadają atrybutom DIANY, wymienionym w Dodatku 1 mojej pracy. Węzeł DIANY zaimplementowany jest jako obiekt typu NODE. Typ rekordowy NODE zawiera jedno pole stałe: 'lx_srcpos' typu 'source_position'. Pole to odpowiada atrybutowi 'lx_srcpos', przechowujacemu pozycję danej konstrukcji Ady w tekście źródłowym programu. Atrybut ten może byc wykorzystany np. do wskazywania dokładnego miejsca wystąpienia błędu. W niniejszej implementacji atrybut ten jest wypełniany podczas odczytywania postaci pośredniej z pliku, nie jest jednak wykorzystywany.

Pole znacznikowe 'kind' części zmiennej rekordu jest typu 'NODE_NAME' i reprezentuje rodzaj danego węzła. Dla każdej wartości typu 'NODE_NAME' został zdefiniowany wariant o jednej składowej, ktorej nazwa składa się z nazwy węzła i przedrostka 'c_'. Składowe te są zdefiniowane jako wskaźniki do rekordów odpowiadajacych danemu rodzajowi węzła.

Cała struktura drzewa DIANY jest zapisana w tablicy 'NodeArray'. Atrybuty reprezentujace klasy należą do typu TREE. Typ ten jest zaimplementowany jako wartość indeksu odpowiadająca danemu węzłowi w tablicy 'NodeArray'.

Ta wewnętrzna struktura jest jednak ukryta przed użytownikiem modułu 'Diana'. Jest to możliwe, gdyż wszystkie operacje na węzłach w innych modułach są wykonywane za pomocą

Dodatek 2. Instrukcje A-kodu.

Dodatek ten zawiera alfabetyczną listę wszystkich instrukcji A-kodu. Każdej instrukcji towarzyszy krótki opis działania, angielski źródłosłów mnemoniki, oraz opis efektu instrukcji na stosie. W lewej kolumnie znajduje się opis stanu stosu przed wykonaniem instrukcji, a w prawej - po wykonaniu instrukcji. Litery oznaczaja wartość danego typu. Szczyt stosu znajduje się z prawej strony. Np. zapis: (a,i) oznacza, że na szczycie stosu znajduje się liczba całkowita, pod szczytem stosu natomiast znajduje się adres.

Jeśli instrukcja wymaga argumentów określonego typu, to pod jej mnemoniką znajduje się lista dopuszczalnych typów. Np. zapis: x:ai oznacza, że dla danej instrukcji argumenty mogą być adresami lub liczbami całkowitymi.

Typy kwalifikujące instrukcje:

a adres (address) - 4 bajty
b logiczny (Boolean) - 1 bajt
c znakowy (character) - 1 bajt
i całkowity (integer) - 2 bajty
s ciąg znaków (string)
x jeden z powyższych (opis działania instrukcji może podać pewne ograniczenia).

Instrukcja:	Efekt na stosie: szczyt stosu przed operacją	po wykonaniu instrukcji
ABO (Abort)	(a)	()
Wykonaj instrukcję abort w stosunku do zadania, którego wskaźnik znajduje się na szczycie stosu.		
ABSx x:i (Absolute)	(x)	(x)
Oblicz wartość bezwzględną liczby znajdującej się na szczycie stosu.		
ACC (Accept)	(i_1,i_2)	()
Wykonaj instrukcję accept. Wartość i_1 określa numer wejścia, i_2 oznacza numer w rodzinie wejść.		
ACA p <etykieta> (Activate Access Task Variable)	(a,i)	(a)
Uaktywnij zadanie. Adres pod szczytem stosu jest wskaźnikiem do bloku opisującego ilość wejść zadania. Blok ten zawiera liczbę wejść oraz pary liczb określające dla każdego wejścia dolną i górną granicę zmian indeksu wejścia traktowanego jak rodzina wejść (jeśli wejście jest pojedyncze, dolna i górna granica są		

równe 1). Liczba na szczycie stosu określa zagłębienie definicji typu wskaźnikowego. Parametr p określa zagłębienie definicji typu zadaniowego, etykieta wskazuje na początek kodu treści zadania. W wyniku wykonania instrukcji na stosie umieszczony zostaje wskaźnik do opisu uaktywnionego zadania.

CT p <etykieta> (a) (a)
(Activate Task)

Uaktywnij zadanie. Adres na szczycie stosu jest wskaźnikiem do bloku opisującego liczbę wejść zadania. Blok ten zawiera liczbę wejść oraz pary liczb określające dla każdego wejścia dolną i górną granicę zmian indeksu wejścia traktowanego jak rodzina wejść (jeśli wejście jest pojedyncze, dolna i górna granica są równe 1). Parametr p określa zagłębienie definicji typu zadaniowego, etykieta wskazuje na początek kodu treści zadania. W wyniku wykonania instrukcji na stosie umieszczony zostaje wskaźnik do opisu uaktywnionego zadania.

DDx (x,x) (x)
:i (Add)

Dodaj dwie liczby znajdujące się na szczycie stosu, wynik umieść na szczycie stosu.

LO q (i) (a)
(Allocate)

Przydziel obszar pamięci o rozmiarze określonym przez liczbę znajdującą się na szczycie stosu. Instrukcja powoduje umieszczenie na stosie adresu przydzielonego obszaru. Jeśli nie ma wystarczającej ilości pamięci zgłaszany jest wyjątek STORAGE_ERROR. Parametr instrukcji określa położenie w statycznej strukturze programu bloku będącego "właścicielem" przydzielonego obszaru. Po zakończeniu wykonywania tego bloku przydzielona pamięć może być automatycznie odzyskana. Parametr równy -1 oznacza, że pamięć jest przydzielona dla bloku otaczającego dynamicznie aktualnie wykonywany blok. Wykorzystywane jest to do przydziału pamięci dla wartości funkcji.

CHR (i) (c)
(Convert Integer to Character)

Zamień liczbę całkowitą znajdującą się na szczycie stosu na znak.

CSP <nazwa> specjalny
(Call Standard Procedure)

Wywołaj standardową procedurę o podanej nazwie.

CSTx q żaden
x:ais (Constant)

Pseudoinstrukcja służąca do generowania stałych.

CUP p q specjalny
(Call User Procedure) + 'RelativeResult'

Wywołaj procedurę zdefiniowaną przez użytkownika. Pierwszy parametr określa rozmiar (w bajtach) obszaru pamięci zajętego przez parametry procedury. Drugi parametr jest etykietą wywoływanej procedury.

DECx q (x) (x)
x:abci (Decrement)

Zmniejszenie wartości na szczycie stosu o wartość będącą argumentem.

DIVx (x_1,x_2) (x)
x:i (Divide)

Podziel liczbę x_1 przez x_2, wynik umieść na szczycie stosu.

DPLx (x) (x,x)

x:abci (Duplicate)
Kopiuje element na szczycie stosu.

EAC () ()
(End of Accept)
Koniec instrukcji accept.

EEX specjalny
(End of Exception Handler)
Koniec segmentu obsługi wyjątków. Instrukcja ta jest wykonywana tylko wtedy, gdy pewien wyjątek nie był obsłużony w danym bloku. Powoduje propagację wyjątku do bloku bezpośrednio otaczającego dany blok.

ENT p <etykieta> specjalny
(Enter Block)
Twórz nową ramkę struktury blokowej programu. Dla p=1 drugi parametr określa rozmiar obszaru zmiennych lokalnych. Dla p=2 drugi parametr określa rozmiar obszaru roboczego ramki.

EQUx (x,x) (b)
x:bci (Equal)
Porównaj dwie wartości na szczycie stosu, ustaw wartość TRUE jeśli są równe, FALSE w przeciwnym wypadku.

ETD q (a,i_1,i_2,i) (b)
(Timed Entry Call)
Wykonaj instrukcję selektywnego wejścia z gałęzią delay (LRM 9.7.3). Na szczycie stosu znajduje się okres czasu, przez który należy próbować wywołać wejście. Pozostałe parametry jak dla instrukcji ETR. Po wykonaniu na stosie umieszczana jest wartość TRUE jeśli doszło do spotkania. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

ETE q (a,i_1,i_2) (b)
(Conditional Entry Call)
Wykonaj instrukcję selektywnego wejścia z gałęzią else (LRM 9.7.2). Parametry jak dla instrukcji ETR. Po wykonaniu na stosie umieszczana jest wartość TRUE jeśli doszło do spotkania. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

ETK specjalny
(End of Task)
Koniec kodu treści zadania.

ETR q (a,i_1,i_2) ()
(Entry Call)
Wywołanie wejścia zadania określonego przez adres znajdujący się pod szczytem stosu. Numer wejścia jest określony przez liczbę i_1, a numer w ramach rodziny przez liczbę znajdującą się na szczycie stosu. Parametr q ma takie samo znaczenie jak pierwszy parametr instrukcji CUP - określa ilość pamięci zajętą przez parametry wejścia. Parametry te umieszczane są jak dla wywołania procedury.

EXC p q specjalny
(Exception)
Obsługa wyjątku. Pierwszy parametr określa obsługiwany wyjątek. Drugi parametr określa etykietę obsługi następnego wyjątku.

EXH <etykieta> żaden
(Exception Handler)
Określa adres startowy segmentu obsługi wyjątków bieżącego bloku. Instrukcja EXH występuje zawsze po deklaracjach, a przed pierwszą instrukcją bloku. Dodatkową funkcją tej instrukcji jest więc określenie końca ciągu instrukcji A-kodu wygenerowanych dla deklaracji bloku. Jest to niezbędne z powodu odmiennego traktowania wyjątków zgłoszonych podczas wykonywania deklaracji

i wyjątków zgłoszonych podczas wykonywania instrukcji danego bloku.

EXL <etykieta> <wyjątek> żaden
(Exception label)
Kojarzy etykietę z konkretnym wyjątkiem. Od tej pory ta etykieta może być używana w instrukcjach EXC, RAI.

FJP <etykieta> (b) ()
(False Jump)
Skok pod wskazany adres, jeśli na szczycie stosu znajduje się wartość FALSE.

FRE (a) ()
(Free)
Zwolnij obszar pamięci przydzielony wcześniej instrukcją ALO.

GEQx (x,x) (b)
x:bci (Greater or Equal)
Ustaw TRUE jeśli wartość znajdująca się bezpośrednio pod szczytem stosu jest większa lub równa wartości na szczycie stosu. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

GET q (a) (...)
(Get onto Stack)
Umieszcza na szczycie stosu q bajtów pobranych spod adresu a.

GREx (x,x) (b)
x:bci (Greater)
Ustaw TRUE jeśli wartość znajdująca się bezpośrednio pod szczytem stosu jest większa od wartosci na szczycie stosu. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

INCx q (x) (x)
x:abci (Increment)
Zwiększenie wartości na szczycie stosu o wartość będącą argumentem.

INDx q (a) (x)
x:abci (Indexed Fetch)
Umieść na stosie wartość znajdującą się pod adresem wyliczonym jako suma adresu znajdującego się na szczycie stosu i przesunięcia będącego argumentem instrukcji.

IXA q (a,i) (a)
(Compute Indexed Address)
Wylicz adres korzystając z adresu bazowego i przesunięcia pomnożonego przez argument instrukcji (współczynnik skalujący).

LAO p q () (a)
(Load Base-level Address)
Umieść na stosie adres zmiennej globalnej.

LCA <etykieta> () (a)
(load Constant Address)
Powoduje umieszczenie na stosie adresu określonego etykietą.

LDA p q () (a)
(Load Address with Level p)
Umieść na stosie adres zmiennej lokalnej określony argumentami instrukcji.

LDCx q () (x)
x:abci (Load Constant)
Umieszcza na stosie stałą, która jest argumentem instrukcji.

LDOx p q () (x)
x:abci (Load Contents of Base-level Address)
Parametr p określa numer jednostki kompilacji (0 oznacza bieżącą jednostkę). Parametr q oznacza przesunięcie względem początku obszaru danej jednostki).

LEQx (x,x) (b)
x:bci (Less or Equal)
Ustaw TRUE jeśli wartość znajdująca się bezpośrednio pod szczytem stosu jest mniejsza lub równa wartości na szczycie stosu. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

LESx (x,x) (b)
x:bci (Less)
Ustaw TRUE jeśli wartość znajdująca się bezpośrednio pod szczytem stosu jest mniejsza od wartosci na szczycie stosu. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

LODx p q () (x)
x:abci (Load Contents of Address)
Umieść na szczycie stosu wartość spod podanego adresu. Parametr p określa statyczne położenie danego obiektu w strukturze blokowej programu. Parametr q określa przesunięcie w ramach danej ramki.

LVB <etykieta> specjalny
(Leave Block)
Powoduje opuszczenie danej liczby zagnieżdżonych bloków. Ponieważ nie zawsze w chwili generowania instrukcji jest znana liczba bloków, które należy opuścić, parametrem jest etykieta. W innym miejscu programu jest określona wartość odpowiadająca tej etykiecie.

MODx (x_1,x_2) (x)
x:i (Modulo)
Wykonaj operację odpowiadającą predefiniowanej operacji mod języka Ada (x := x_1 mod x_2).

MOV q (a,a) ()
(Move)
Prześlij ilość bajtów określona przez argument instrukcji spod adresu będącego na szczycie stosu pod adres znajdujący się pod wierzchołkiem stosu.

MVV (a_1,a_2,i) ()
(Move Variable Number of Bytes)
Prześlij ilość bajtów określona przez liczbę, znajdującą się na szczycie stosu spod adresu a_2 pod adres a_1. Kopiowanie musi działać poprawnie również gdy obszary: źródłowy i docelowy nakładają się (LRM 5.2.1).

MST p q specjalny
(Mark Stack)
Przygotowuje stos do wywołania procedury lub funkcji. Parametr q określa względne zanurzenie procedury. Parametr p podaje rozmiar wartości zwracanej przez wykonywany podprogram. Dla procedur jest więc równy 0.

MULx (x,x) (x)
x:i (Multiply)
Pomnóż dwie liczby znajdujące się na szczycie stosu, wynik umieść na szczycie stosu.

NEGx (x) (x)
x:i (Negate)
Zmień znak liczby na przeciwny.

NEQx (x,x) (b)
x:bci (Not Equal)
Ustaw TRUE jeśli wartosc na szczycie stosu jest różna od wartości znajdującej się bezpośrednio pod szczytem stosu. W przeciwnym wypadku na stosie umieszczana jest wartość FALSE.

NOT (b) (b)

(Boolean not)
Zaneguj logicznie wartość na szczycie stosu.

PKB <nazwa pakietu>
(Package Body)
Występuje na początku treści pakietu, będącej jednostką kompilacji.

PKG <nazwa pakietu>
(Package)
Występuje na początku pakietu, będącego jednostką kompilacji.

PRO <nazwa procedury>
(Procedure)
Występuje na początku procedury, będącej jednostką kompilacji.

PUT q (a,...) ()
(Put from Stack)
Umieść q bajtów ze szczytu stosu pod adresem a.

RAI [<etykieta>] specjalny
(Raise)
Powoduje zgłoszenie wyjątku określonego przez parametr instrukcji. Bezparametrowa instrukcja RAI jest odpowiednikiem bezparametrowej instrukcji raise Ady.

REMx (x,x) (x)
x:i (Remainder)
Wykonaj operację odpowiadającą predefiniowanej operacji rem języka Ada.

RET q specjalny
(Return from Procedure/Function)
Powrót z procedury lub funkcji. Parametr określa rozmiar obszaru parametrów podprogramu (łącznie z obszarem określonym przez stałą 'RelativeResultOffset').

RFL <etykieta> żaden
(Reference to Label)
Definiuje etykietę procedury zadeklarowanej w pakiecie. Występuje w treści pakietu będącego jednostką kompilacji oraz w tych jednostkach kompilacji, w których w klauzuli with jest wymieniony dany pakiet.

RFP q <nazwa pakietu> żaden
(Reference to Package)
Jeśli parametr q jest różny od zera, definiuje numer pakietu występującego w klauzuli with danej jednostki kompilacji. Numer ten jest wykorzystywany następnie jako parametr instrukcji: SRO i LDO. Jeśli jednostką biblioteczną wymienioną w klauzuli with jest procedura, parametr q ma zerową wartość

SLA q $(i_1, i_2, \ldots i_{2q})$ (i)
(Select)
Wykonaj instrukcję select (LRM 9.7.1), bez opcji else, delay, terminate. Parametr instrukcji określa ilość wszystkich opcji kompilowanej instrukcji select. Przed wykonaniem instrukcji na stosie umieszczane są pary liczb określające numery wejść i numery w ramach rodziny wejść (jak dla instrukcji ACC). Opcje, dla których warunek when (o ile wystąpił) nie jest spełniony, mają jako numer wejścia podaną liczbę 0. Instrukcja powoduje zawieszenie zadania, aż któreś z tych wejść zostanie wywołane przez inne zadanie i umieszczenie na stosie numeru wybranej opcji (opcje są numerowane od 0).

SLD q $(i_1, i_2, \ldots i_{2q})$ (i)
(Select with Delay Alternative)
Wykonaj instrukcję select (LRM 9.7.1), z gałęzią delay. Przed wykonaniem instrukcji na stosie umieszczane są opisy opcji

instrukcji select. Opcje zawierające instrukcję delay mają jako numer wejścia podaną liczbę -2. Zamiast numeru w ramach rodziny podany jest wtedy parametr instrukcji delay. Opis opcji accept składa się z pary liczb oznaczających numer wejścia i numeru w ramach wejścia, jeśli warunek when nie jest spełniony, jako numer wejścia podawana jest liczba zero. Instrukcja powoduje zawieszenie zadania aż któreś z wejść zostanie wywołane przez inne zadanie lub upłynie czas, określony przez instrukcje delay o minimalnej wartości parametru i umieszczenie na stosie numeru wybranej opcji.

SLE q $(i_1, i_2, \ldots i_{2q})$ (i)

(Select with Else Alternative)

Wykonaj instrukcję select (LRM 9.7.1), z gałęzią else. Przed wykonaniem instrukcji na stosie umieszczane są parametry takie same jak dla instrukcji SLA. Instrukcja powoduje sprawdzenie, czy któraś z otwartych opcji zawiera obsługę wejścia jest wywoływanego przez inne zadanie; jeśli tak, to numer tej opcji jest umieszczany na stosie, w przeciwnym wypadku na stosie umieszczana jest liczba -1.

SLT q $(i_1, i_2, \ldots i)$ (i)

(Select with Terminate Alternative)

Wykonaj instrukcję select (LRM 9.7.1), z gałęzią terminate. Przed wykonaniem instrukcji na stosie umieszczane są parametry takie same jak dla instrukcji SLA. Instrukcja powoduje sprawdzenie, czy któraś z otwartych opcji zawiera obsługę wejścia wywoływanego przez inne zadanie; jeśli tak, to numer tej opcji jest umieszczany na stosie, w przeciwnym wypadku zadanie jest kończone (o ile są spełnione warunki zakończenia zadania- LRM 9.4).

SROx p q (x) ()

x:abci (Store at Base Level Address)

Prześlij zawartość szczytu stosu do zmiennej globalnej. Parametr p określa numer pakietu (zdefiniowany instrukcją RFP). Wartość zero oznacza bieżący pakiet.

STOx (a,x) ()

(Store)

x:abci

Prześlij zawartość szczytu stosu pod adres znajdujący się pod szczytem stosu.

STRx p q (x) ()

x:abci (Store)

Przesłanie zawartości szczytu stosu pod wskazany adres.

SUBx (x_1, x_2) (x)

x:i (Subtract)

Odejmij liczbę x_2 od liczby x_1, wynik umieść na szczycie stosu.

SWPx (x_1, x_2) (x_2, x_1)

x:abci (Swap)

Zamień miejscami dwa obiekty na szczycie stosu.

TJP <etykieta> (b) ()

(True Jump)

Skok pod wskazany adres, jeśli na szczycie stosu znajduje sie wartość TRUE.

UJP <etykieta> żaden

(Unconditional Jump)

Skok bezwarunkowy pod wskazany adres.

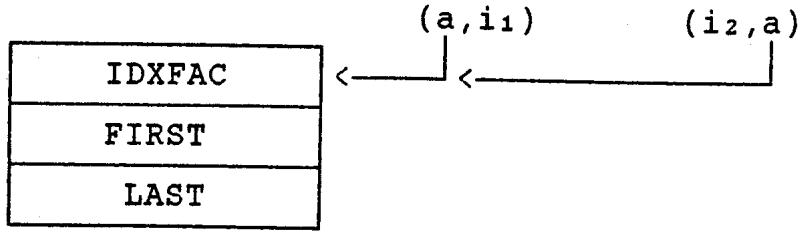
XJP q (i) ()

(Indexed Jump)

Wykonaj skok według tablicy skoków. Jeśli wartość na szczycie stosu jest nieujemna i mniejsza od argumentu instrukcji, to wykonaj skok o tym numerze z tablicy skoków, znajdującej się bezpośrednio za instrukcją XJP. W przeciwnym wypadku wykonaj skok o numerze będącym parametrem instrukcji.

Dodatek 3. Standardowe procedury A-kodu.

AR1



(Array Indexing 1)

Procedura służąca do obliczenia przesunięcia od początku obszaru zajmowanego przez tablicę, dla danego wymiaru. Adres a wskazuje na opis danego wymiaru. Liczba i_1 jest wartością indeksującą. Jeśli spełniony jest warunek:

$$FIRST <= i_1 <= LAST,$$

obliczana jest wartość i_2:

$$i_2 := IDXFAC * (i_1 - FIRST),$$

w przeciwnym wypadku zgłaszany jest wyjątek CONSTRAINT_ERROR.

AR2 (a,i_1) (i_2)

(Array Indexing 2)

Działa jak procedura ARR1. Jedyną różnicą jest to, że po jej poprawnym wykonaniu na stosie nie jest pozostawiany wskaźnik do opisu wymiaru. Jest wykorzystywana do obliczania przesunięcia dla ostatniego wymiaru tablicy.

CLB (a) (b)

(CALLABLE)

Przed wykonaniem procedury na stosie znajduje się wskaźnik do opisu zadania. Procedura zwraca wartość atrybutu CALLABLE dla tego zadania.

CLN (a,a) ()

(Check length of slice)

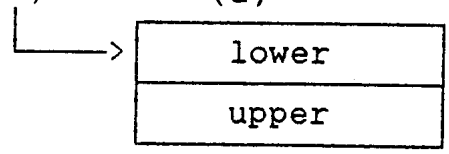
Przed wykonaniem procedury na stosie znajdują się dwa wskaźniki do opisów zakresu. Jeśli długości tych zakresów są różne, zgłaszany jest wyjątek CONSTRAINT_ERROR. W przeciwnym razie nic nie jest robione.

CNT (a,i_1,i_2) (i)

(COUNT)

Przed wykonaniem procedury na stosie znajduje się wskaźnik do opisu zadania. Liczba i_1 określa numer wejścia, a i_2 - numer w ramach rodziny. Procedura zwraca wartość atrybutu COUNT dla tego wejścia.

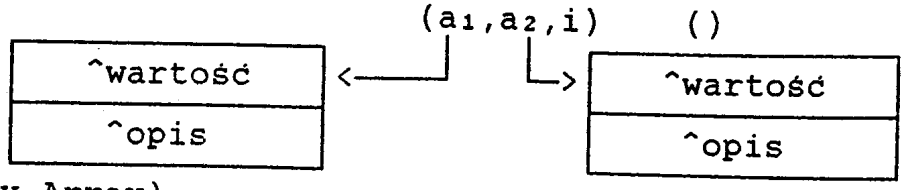
CVB (i,a) (i)



(Check Variable Bounds)

Sprawdź, czy liczba znajdująca się pod szczytem stosu jest w zakresie, określonym przez adres na szczycie stosu. Jeśli tak (tj. spełniony jest warunek (lower <= i <= upper), to na szczycie stosu pozostaje ta liczba. W przeciwnym wypadku- zgłaszany jest wyjątek CONSTRAINT_ERROR.

CYA



(Copy Array)

Powoduje skopiowanie tablicy, która jest wskazywana przez a_2 do tablicy wskazywanej przez a_1. Wartość na szczycie stosu określa liczbę wymiarów kopiowanych tablic (obie tablice mają zawsze tą samą liczbę wymiarów). Gdy zakresy obu tablic dla któregoś z wymiarów nie są zgodne, zgłaszany jest wyjątek CONSTRAINT_ERROR.

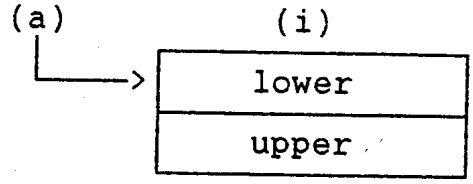
LBD

(i,a) (i)

(Lower bound)

Na szczycie stosu znajduje się wskaźnik do opisu zakresu (patrz procedura LEN) oraz liczba całkowita. Jeśli ta liczba znajduje się poza podanym zakresem, zgłaszany jest wyjątek CONSTRAINT_ERROR. W przeciwnym razie na szczycie stosu umieszczana jest wartość (i - lower).

LEN



(Length)

Przed wykonaniem procedury na stosie znajduje się wskaźnik do opisu zakresu (pary liczb określających odpowiednio minimalną i maksymalną wartość zakresu). Jako efekt działania procedura umieszcza na stosie liczbę (upper - lower + 1), gdy zachodzi warunek (lower <= upper) lub liczbę 0 w przeciwnym wypadku.

PUA

(a_1,a_2,a_3,i) ()

(Put Array)

Działa jak procedura CYA. Różnicą jest to, że opis drugiej tablicy (adresy a_2 i a_3) umieszczony jest na stosie, a nie wskazywany przez adres.

TRM

(a) (b)

(TERMINATED)

Przed wykonaniem procedury na stosie znajduje się wskaźnik do opisu zadania. Procedura zwraca wartość atrybutu TERMINATED dla tego zadania.

Dodatek 5. Wybrane przebiegi testowania.

Dodatek ten zawiera przykładowe przebiegi testowe modułu generatora A-kodu. W rzeczywistości zestawów danych testowych było więcej. Przygotowane dane testowe powodowały przynajmniej jednokrotne wykonanie każdej instrukcji projektowanego modułu. Jedynym wyjątkiem było nie wykonywanie wszystkich opcji instrukcji case gdy poszczególne opcje miały bardzo krótki i prawie identyczny kod. W Dodatku znajduje się osiem jednostek kompilacji stanowiących dane testowe i wygenerowany dla nich kod. Dla wygody osób, które będą chciały skorzystać również z pozostałych (nie zamieszczonych tu) danych testowych, zachowana została oryginalna numeracja testowanych jednostek kompilacji.

Pakiet Test22 zawiera deklarację typu wyliczeniowego 'ENUM' i jednej zmiennej tego typu. Procedura Test23 korzystająca z pakietu Test22 zawiera instrukcję pętli for, której zmienna robocza przyjmuje kolejno wszystkie wartości typu 'ENUM'.

Pakiet Test12 demonstruje sposób kompilowania instrukcji przypisania oraz procedury wykorzystującej parametry skalarne wszystkich trybów. Procedura Test13 zawiera przykład wywołania procedury zadeklarowanej w innej jednostce kompilacji.

Procedura Test21 demonstruje sposób zgłaszania i obsługi wyjątków.

Pakiet Test4 zawiera deklarację typu tablicowego i jednej zmiennej tego typu. Procedura Test26 pokazuje sposób kompilacji deklaracji i wywołania funkcji, zwracającej wartość typu tablicowego. Przy okazji pokazany jest sposób dostępu do zmiennych zadeklarowanych w innej jednostce kompilacji.

Dwa pierwsze przykłady są kompletne, tzn. zawierają, oprócz postaci źródłowej w języku Ada, pełen zapis kompilowanej jednostki w postaci DIANY. Pozostałe przykłady składają się z wersji źródłowej oraz jej odpowiednika w A-kodzie.

Generator kodu po niektórych instrukcjach A-kodu drukuje komentarze wskazujące związek między danym fragmentem A-kodu a tekstem źródłowym. Dodatkowo pojawiające się po raz pierwszy nieoczywiste sekwencje A-kodu skomentowano w Dodatku obszerniej.

Tekst źródłowy pakietu:

```
package Test22 is
  type ENUM is ( 'A', PL_A, 'E', PL_E);
  E1 : ENUM;
end Test22;
```

Reprezentacja w postaci DIANY:

```
D1001 : compilation          [
                               as_list < D1002^ > ]
D1002 : comp_unit            [
                               as_context   D1003^;
                               as_unit_body D1005^;
                               as_pragma_s  D1004^ ]
D1003 : context              [
                               as_list < > ]
D1004 : pragma_s             [
                               as_list < > ]
D1005 : package_decl         [
                               as_id           D1006^;
                               as_package_def D1007^ ]
D1006 : package_id          [
                               lx_symrep   "TEST22";
                               sm_spec     D1007^;
                               sm_body     VOID;
                               sm_address  VOID;
                               sm_stub     VOID;
                               sm_first    D1006^; ]
D1007 : package_spec         [
                               as_decl_s1  D1008^;
                               as_decl_s2  D1009^ ]
D1008 : decl_s               [
                               as_list     < D1010^ D1020^ > ]
D1009 : decl_s               [
                               as_list     <  >  ]
D1010  : type                [
                               as_id           D1011^;
                               as_dscrmt_var_s D1012^;
                               as_type_spec    D1013^ ]
D1011  : type_id             [
                               lx_symrep    "ENUM";
                               sm_type_spec D1013^;
                               sm_first     D1011^ ]
D1012  : dscrmt_var_s        [
                               as_list < > ]
D1013  : enum_literal_s      [
                               as_list < D1014^ D1015^
                                         D1016^ D1017^ >
                               sm_size VOID ]
D1014  : def_char            [
                               lx_symrep     "'A'";
                               sm_obj_type   D1013^;
                               sm_pos         0;
                               sm_rep         0 ]
D1015  : enum_id             [
                               lx_symrep      "PL_A";
                               sm_obj_type    D1013^;
                               sm_pos         1;
```

```
                                        sm_rep          1 ]
D1016  : def_char                   [
                                        lx_symrep      "'E'";
                                        sm_obj_type    D1013^;
                                        sm_pos          2;
                                        sm_rep          2 ]
D1017  : enum_id                    [
                                        lx_symrep       "PL_E";
                                        sm_obj_type     D1013^;
                                        sm_pos          3;
                                        sm_rep          3 ]
D1020 : var                         [
                                        as_id_s        D1021^;
                                        as_type_spec   D1013^;
                                        as_object_def  VOID ]
D1021 : id_s                        [
                                        as_list < D1022^ > ]
D1022 : var_id                      [
                                        lx_symrep    "E1";
                                        sm_obj_type  D1013^;
                                        sm_address   VOID;
                                        sm_obj_def   VOID ]
```

Wygenerowany A-kod:

```
 PKG        TEST22
 RFP        1       STANDARD
 EXL        L       2       CONSTRAINT_ERROR
 EXL        L       3       NUMERIC_ERROR
 EXL        L       4       PROGRAM_ERROR
 EXL        L       5       STORAGE_ERROR
 EXL        L       6       TASKING_ERROR
L       1                       -- etykieta startowa
 ENT        1       L       7
 ENT        2       L       8
 EXH        L       9
                                -- część akcyjna jest pusta
 RET        2
L       9
 EEX
L       7       =       2
L       8       =       2
 Q
```

Tekst źródłowy:

```
with Test12, Test12; use Test12;
procedure Test13 is
  M, N: INTEGER;
begin
  P(77, M, N);
end Test13;
```

Wygenerowany A-kod:

```
 PRO        TEST13
 RFP        1       STANDARD
 EXL        L       2       CONSTRAINT_ERROR
 EXL        L       3       NUMERIC_ERROR
 EXL        L       4       PROGRAM_ERROR
 EXL        L       5       STORAGE_ERROR
 EXL        L       6       TASKING_ERROR
 RFP        2       TEST12
 RFL        L       7                       -- procedure P
L     1                                      -- procedure TEST13
 ENT        1       L       9
 ENT        2       L      10
 EXH        L      11                       -- begin
 MST        0       1
 LDCI      77
 LDA        0       0                       -- M
 DPLA
 INDI       0
 LDA        0      -2                       -- N
 DPLA
 INDI       0
 CUP       14       L       7               -- P
L     8                                      -- copy out
 RET        2
L    11
 EEX
L     9     =       4
L    10     =      24
 Q                                           -- end TEST13
```

Tekst źródłowy:

```
procedure Test21 is
  ERROR: exception;
begin
  null;
exception
  when NUMERIC_ERROR =>
    raise;
  when others =>
    raise ERROR;
end Test21;
```

Wygenerowany A-kod:

```
 PRO        TEST21
 RFP        1       STANDARD
 EXL        L       2       CONSTRAINT_ERROR
 EXL        L       3       NUMERIC_ERROR
 EXL        L       4       PROGRAM_ERROR
 EXL        L       5       STORAGE_ERROR
 EXL        L       6       TASKING_ERROR
L     1                                -- procedure TEST21
 ENT        1       L       8
 ENT        2       L       9
 EXL        L      10       ERROR
 EXH        L      11                -- begin
-- null
L     7                                -- copy out
 RET        2
L    11
 EXC        L       3       L      13 -- NUMERIC_ERROR
L    12
 RAI
 UJP        L       7
L    13
L    14
 RAI        L      10                -- ERROR
 UJP        L       7
L     8     =       0
L     9     =       0
 Q                                    -- end TEST21
```

Bibliografia

[1] DIANA. An Intermediate Language for Ada.
Edited by G.Goos, W.A.Wulf, A.Evans,Jr., and K.J.Butler
Springer-Verlag, Berlin Heidelberg, 1983

[2] Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815A-1983
US DoD, February 17, 1983

[3] Pemberton S. and Daniels M.
Pascal Implementation: The P4 Compiler.
Ellis Horwood Ltd., Chichester, 1983

[4] Waite W. M., Goos G.
Konstrukcja kompilatorów.
WNT, Warszawa 1989

[5] Paprocki A.
Problemy implementacji języka Ada.
Informatyka, 1990, nr 1

[6] Jansohn H.S., Holzapfel R.
Der Einsatz von Ada in der Praxis. Compiler-Bau.
Chip Plus, 1989, nr 3

[7] Nestor J.R., Wulf W.A., Lamb D.A.
IDL - Interface Description Language.
Carnegie-Mellon University, 1981