Asembler. Sztuka programowania

Autor: Randall Hyde
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-602-0
Tytuł oryginału: The Art of Assembly Language
Format: B5, stron: 968

Książka „Asembler. Sztuka programowania" to podręcznik programowania w języku asemblera oparty na języku HLA. Opisuje 32-bitową architekturę procesorów Intel oraz zasady tworzenia programów w języku HLA. Przedstawia sposoby pisania, kompilacji i uruchamiania programów wykorzystujących różne, nawet najbardziej złożone typy danych.

- Wprowadzenie do języka HLA
- Sposoby reprezentacji danych
- Organizacja pamięci i tryby adresowania
- Typy danych
- Podział programu na procedury i moduły
- Sterowanie wykonaniem programu
- Instrukcje arytmetyczne
- Operacje na plikach
- Operacje bitowe i łańcuchowe
- Makrodefinicje
- Klasy i obiekty
- Połączenie asemblera z programami w innych językach

Przekonaj się, jak prosty jest język asemblera

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

Spis treści

1.9. Podstawowe struktury sterujące wykonaniem programu HLA

Instrukcje `mov`, `add` oraz `sub`, choć niezwykle użyteczne, nie wystarczają do napisania sensownych programów. Aby takie programy mogły powstać, podstawowe instrukcje muszą zostać uzupełnione o podstawowy zbiór struktur sterujących wykonaniem kodu umożliwiających tworzenie pętli i podejmowanie decyzji w programie. Język HLA przewiduje szereg wysokopoziomowych struktur sterujących wykonaniem charakterystycznych dla języków wyższego niż asembler poziomu. Programista HLA ma więc do dyspozycji konstrukcje `if..then..elseif..else..endif`, `while..endwhile`, `repeat..until` i tak dalej. Dopiero znajomość i umiejętność wykorzystywania owych konstrukcji pozwala na stworzenie prawdziwego programu.

Przed zagłębieniem się w cechach struktur sterujących należałoby podkreślić, że nie mają one odpowiedników w zestawie instrukcji maszynowych procesorów 80x86. Kompilator HLA tłumaczy owe struktury do postaci szeregu prawdziwych instrukcji maszynowych za programistę. Sposób, w jaki to czyni, opisany zostanie w dalszej części książki; wtedy też Czytelnik nauczy się pisać kod w czystym języku asemblerowym, bez wykorzystania wysokopoziomowych cech kompilatora HLA. Na razie jednak, z racji niewielkiego jeszcze zasobu wiedzy o języku asemblerowym, nieco sztuczne struktury sterujące HLA będą bardzo pomocne.

Warto przy tym pamiętać, że wysokopoziomowe na pierwszy rzut oka struktury sterujące nie są w istocie tak wysokopoziomowe. Mają one bowiem jedynie ułatwić rozpoczęcie pisania programów w języku asemblerowym — w żadnym razie nie zastępują asemblerowych instrukcji sterujących wykonaniem kodu. Już wkrótce okaże się więc, że owe wysokopoziomowe struktury sterujące mają szereg ograniczeń, które w miarę nabierania przez Czytelnika doświadczenia i w miarę komplikowania kolejnych programów staną się dość uciążliwe. To zresztą efekt zamierzony — po osiągnięciu pewnego poziomu umiejętności programista sam stwierdzi, że w wielu przypadkach wygoda (i czytelność programu) wynikająca z zastosowania struktur sterujących HLA nie rekompensuje utraty efektywności, charakterystycznej dla kodu wykorzystującego wprost instrukcje maszynowe procesora.

Niniejsze omówienie kierowane jest do tych Czytelników, którzy znają już przynajmniej jeden z popularnych języków wysokiego poziomu. Znajomość ta znakomicie uprości prezentację struktur sterujących wykonaniem programu HLA, pozwalając na pominięcie opisów sposobów zastosowania owych struktur w typowych zadaniach programistycznych. Czytelnicy potrafiący korzystać ze struktur sterujących wykonaniem programu w dowolnym z języków wysokiego poziomu nie powinni być zaskoczeni, kiedy odkryją, że analogiczne struktury wykorzystywane są w programach HLA w sposób dosłownie identyczny.

1.9.1. Wyrażenia logiczne w instrukcjach HLA

Niektóre z instrukcji sterujących wykonaniem programu HLA wymagają określenia w miejsce operandu wartości logicznej („prawda" bądź „fałsz"); ścieżka wykonania programu zależy wtedy od wartości wyrażenia logicznego. Przykładami struktur wymagających określenia wyrażenia logicznego są instrukcje `if`, `while` oraz `repeat..until`. Pierwszą oznaką ograniczeń struktur sterujących wykonaniem programu HLA jest składnia owych wyrażeń. W tym jednym miejscu nawyki wyniesione z języków wysokiego poziomu zwracają się przeciwko programiście asemblera: w języku HLA wyrażenia logiczne są bardzo ograniczone — nie sposób konstruować tu wyrażeń równie wymyślnych, z jakimi może mieć do czynienia programista języka C++.

Wyrażenia logiczne w języku HLA mogą przyjąć jedną z następujących postaci[1]:

```
znacznik
!znacznik
rejestr
!rejestr
zmienna-logiczna
!zmienna-logiczna
pamięć-rejestr operator-relacji pamięć-rejestr-stała
rejestr in stała(granica-dolna)..stała(granica-górna)
rejestr not in stała(granica-dolna)..stała(granica-górna)
```

Znacznik może zostać określony za pośrednictwem jednego z symboli wyliczonych w tabeli 1.2.

Tabela 1.2. *Symbole reprezentujące znaczniki*

Symbol	Znacznik	Znaczenie
`@c`	Przeniesienie	„Prawda" dla ustawionego (1) znacznika przeniesienia; „fałsz" dla wyzerowanego (0) znacznika przeniesienia.
`@nc`	Brak przeniesienia	„Prawda" dla wyzerowanego (0) znacznika przeniesienia; „fałsz" dla ustawionego (1) znacznika przeniesienia.
`@z`	Zero	„Prawda" dla ustawionego (1) znacznika zera; „fałsz" dla wyzerowanego (0) znacznika zera.
`@nz`	Brak zera	„Prawda" dla wyzerowanego (0) znacznika zera; „fałsz" dla ustawionego (1) znacznika zera.
`@o`	Przepełnienie	„Prawda" dla ustawionego (1) znacznika przepełnienia; „fałsz" dla wyzerowanego (0) znacznika przepełnienia.
`@no`	Brak przepełnienia	„Prawda" dla wyzerowanego (0) znacznika przepełnienia; „fałsz" dla ustawionego (1) znacznika przepełnienia.
`@s`	Znak	„Prawda" dla ustawionego (1) znacznika znaku; „fałsz" dla wyzerowanego (0) znacznika znaku.
`@ns`	Brak znaku	„Prawda" dla wyzerowanego (0) znacznika znaku; „fałsz" dla ustawionego (1) znacznika znaku.

[1] Istnieje jeszcze kilka postaci dodatkowych; zostaną one omówione w dalszych podrozdziałach i rozdziałach.

Wykorzystanie tych znaczników w wyrażeniach logicznych to już dość zaawansowane programowanie w asemblerze. Sposób konstruowania wyrażeń logicznych angażujących znaczniki słowa stanu procesora zaprezentowany zostanie w następnym rozdziale.

W przypadku, kiedy wartość logiczna angażuje operand rejestrowy, operand ten może odnosić się do zarówno do rejestru 8-bitowego, 16-bitowego, jak i 32-bitowego. Wyrażenie takie przyjmuje wartość „fałsz", jeśli rejestr zawiera wartość zero. Dla każdej innej wartości rejestru wyrażenie logiczne przyjmuje wartość „prawda".

Jeśli w wyrażeniu logicznym uwikłana jest zmienna przechowywana w pamięci, jej wartość jest sprawdzana, a wyrażenie przyjmuje wartość logiczną „prawdy" bądź „fałszu" w zależności od tej wartości, według reguł identycznych jak dla rejestrów. Należy jednak pamiętać, że w wyrażeniu logicznym powinna występować zmienna typu logicznego (`boolean`). Uwikłanie w wyrażeniu logicznym zmiennej innego typu spowoduje błąd. W przypadku potrzeby określenia wartości logicznej zmiennej typu innego niż `boolean` należy skorzystać z wyrażeń logicznych ogólnego przeznaczenia, omówionych poniżej.

Najbardziej ogólna postać wyrażenia logicznego w języku HLA pozwala na określenie operatora relacji wraz z dwoma operandami. Listę dozwolonych kombinacji rodzajów operandów i operatorów relacji zawiera tabela 1.3.

Tabela 1.3. *Dozwolone wyrażenia logiczne*

Operand lewy	Operator relacji	Operand prawy
Zmienna w pamięci bądź rejestr	= albo ==	Zmienna w pamięci, rejestr bądź stała
	<> albo !=	
	<	
	<=	
	>	
	>=	

Nie można określić obu operandów jako operandów pamięciowych. **Lewy operand** można więc utożsamić z operandem źródłowym, a **operand prawy** z operandem docelowym i stosować kombinacje rodzajów operandów takie jak dla instrukcji `add` czy `sub`.

Kolejna analogia operatorów relacji do instrukcji `sub` i `add` objawia się w wymogu identycznego rozmiaru operandów. I tak, oba operandy muszą być albo bajtami, albo słowami, albo podwójnymi słowami. Jeśli operand prawy jest stałą, jego wartość musi nadawać się do zapisania w operandzie lewym.

I jeszcze jedno: jeśli lewy operand jest rejestrem, a prawy to stała dodatnia albo inny rejestr, porównanie odbywa się bez uwzględnienia znaku operandów. Efekty takiego porównania omówione zostaną w następnym rozdziale; na razie wystarczy zapamiętać, że nie powinno się porównywać wartości ujemnej przechowywanej w rejestrze z wartością stałą albo zawartością innego rejestru. Wynik porównania może bowiem odbiegać od oczekiwań programisty.

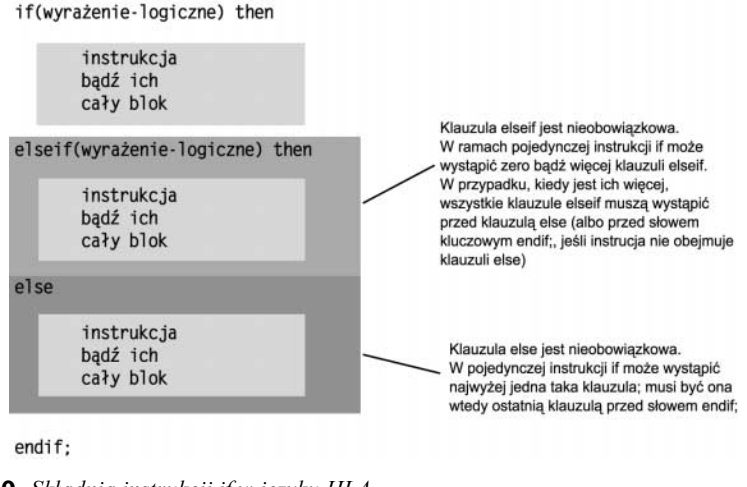
Operatory `in` i `not in` pozwalają na wykonanie testu zawierania się wartości przechowywanej w rejestrze w określonym zbiorze liczb. Na przykład wyrażenie `eax in 2000..20099` daje wartość logiczną `true`, jeśli wartość przechowywana w rejestrze EAX to wartość z zakresu od 2000 do 2099 (włącznie). Operator `not in` ma działanie odwrotne — da wartość logiczną `true` jedynie wtedy, kiedy zawartość rejestru to liczba spoza określonego zbioru. Na przykład wyrażenie `al not in 'a'..'z'` będzie miało wartość `true` tylko wtedy, kiedy znak przechowywany w rejestrze AL nie będzie małą literą alfabetu.

Oto kilka przykładów wyrażeń logicznych w języku HLA:

```
@c
Bool_var
al
ESI
EAX < EBX
EBX > 5
i32 < -2
i8 > 128
al < i8
eax in 1..100
ch not in 'a'..'z'
```

1.9.2. Instrukcje if..then..elseif..else..endif języka HLA

Składnia instrukcji warunkowego wykonania kodu `if` w wydaniu języka HLA prezentowana jest na rysunku 1.10.



Rysunek 1.10. *Składnia instrukcji if w języku HLA*

Wyrażenia występujące w instrukcji `if` muszą odpowiadać jednej z postaci wyrażeń logicznych, zaprezentowanych w poprzednim ustępie. Jeśli wyrażenie przyjmie wartość logiczną „prawda", wykonany zostanie kod umieszczony za klauzulą `then`. W przeciwnym przypadku sterowanie przekazane zostanie do kodu w ramach klauzuli `elseif` bądź `else`.

Ponieważ klauzule `elseif` i `else` nie są obowiązkowe, instrukcja `if` w najprostszej postaci składa się z wyrażenia logicznego i pojedynczej klauzuli `then`, za którą występuje instrukcja lub blok instrukcji wykonywanych warunkowo, w zależności od wartości wyrażenia logicznego:

```
if( eax = 0 ) then

    stdout.put( "blad: wartosc NULL", nl );

endif;
```

Jeśli w trakcie wykonania programu wyrażenie logiczne w instrukcji `if` będzie miało wartość `true`, wykonany zostanie kod znajdujący się pomiędzy słowami `then` i `endif`. W przypadku kiedy wyrażenie logiczne będzie miało wartość `false`, kod ów zostanie pominięty.

Inną powszechnie występującą formą instrukcji `if` jest forma z pojedynczą klauzulą `else`. Instrukcja taka wygląda następująco:

```
if( eax = 0 ) then

    stdout.put( "blad: wartocs NULL wskaznika", nl );

else

    stdout.put( "Poprawna wartosc wskaznika", nl );

endif;
```

Jeśli wyrażenie logiczne przyjmie wartość `true`, wykonany zostanie kod zawarty między słowami `then` a `else`. W przypadku wartości `false` sterowanie przekazane zostanie do kodu ograniczonego klauzulą `else` i słowem `endif`.

Przez osadzanie opcjonalnych klauzul `elseif` w instrukcji `if` można rozbudowywać podejmowanie decyzji o wykonaniu kodu. Na przykład, jeśli rejestr CH zawiera wartość znakową, można na podstawie jego wartości dokonać wyboru spośród dostępnych opcji menu, konstruując następującą instrukcję `if`:

```
if( ch = 'a' ) then

    stdout.put( "Wybrano pozycje menu: 'a'", nl );

elseif( ch = 'b' ) then

    stdout.put( "Wybrano pozycje menu: 'b'", nl );

elseif( ch = 'c' )

    stdout.put( "Wybrano pozycje menu: 'c'", nl );

else

    stdout.put( "Blad: wybor opcji spoza menu", nl );
endif;
```

Choć nie widać tego w powyższym przykładzie, język HLA nie wymaga umieszczania po sekwencji klauzul `elseif` klauzuli `else`. Jednak przy podejmowaniu decyzji o wykonaniu jednej z wielu ścieżek kodu warto uzupełnić instrukcję `if` klauzulą `else` rozpoczynającą ścieżkę kodu obsługi ewentualnych błędów wyboru. Nawet jeśli pozornie wykonanie owego kodu wydaje się niemożliwe, to należy wziąć pod uwagę ewentualny rozwój aplikacji — w kolejnych wersjach kodu wybór ścieżki wykonania może mieć inny przebieg; warto, aby wtedy kod decyzyjny uzupełniony został o obsługę sytuacji wyjątkowych i niespodziewanych.

1.9.3. Iloczyn, suma i negacja w wyrażeniach logicznych

Na liście operatorów zaprezentowanej w poprzednich punktach zabrakło bardzo ważnych operatorów logicznych: operatora iloczynu logicznego (AND), logicznej sumy (OR) oraz logicznej negacji (NOT). Ich zastosowanie w wyrażeniach logicznych omówione zostanie tutaj — wcześniej omówienie takie nie miałoby racji bytu z uwagi na konieczność poprzedzenia go omówieniem instrukcji warunkowego wykonania kodu `if`. Bez możliwości wybiórczego wykonywania kodu trudno zaprezentować realistyczne przykłady zastosowania operatorów logicznych.

W języku HLA operator iloczynu logicznego ma postać znaków `&&`. Jest to operator binarny i jako taki wymaga określenia dwóch operandów; operandy te muszą być poprawnymi wyrażeniami czasu wykonania. Operator ten zwraca wartość logiczną „prawda” wtedy, kiedy oba operandy mają wartość „prawda”. Oto przykład:

```
if( eax > 0 && ch 'a' ) then

    mov( eax, ebx );
    mov( ' ', ch );

endif;
```

Obie instrukcje `mov` zostaną wykonane jedynie wtedy, kiedy równocześnie rejestr EAX będzie miał wartość większą od 0, a rejestr CH będzie zawierał znak „a”. Jeśli którykolwiek z tych warunków nie będzie spełniony, obie instrukcje `mov` zostaną pominięte.

Warto pamiętać, że operandy operatora `&&` mogą być dowolnymi poprawnymi wyrażeniami logicznymi języka HLA — nie muszą to być wyłącznie operatory relacji. Oto kilka poprawnych logicznych z operatorem iloczynu logicznego:

```
@z && al in 5..10
al in 'a'..'z' && ebx
boolVar && !eax
```

W języku HLA wykorzystywana jest metoda **skróconego szacowania** wartości wyrażeń operandów operatora logicznego `&&`. Otóż jeśli lewy operand ma wartość „fałsz”, to operand prawy nie jest już sprawdzany — wyrażenie logiczne natychmiast otrzymuje wartość „fałsz” (co jest jak najbardziej poprawne, bo operator `&&` daje wartość „prawda” wyłącznie dla dwu operandów o wartości „prawda”). Stąd w ostatnim z zaprezentowanych wyżej wyrażeń najpierw sprawdzona zostanie wartość logiczna zmiennej `boolVar`; jeśli będzie to wartość `false`, test wartości logicznej rejestru EAX zostanie pominięty.

Nie sposób nie zauważyć, że wyrażenie z operatorem &&, np. `eax < 10 && ebx <> eax`, jest wyrażeniem logicznym (zwraca albo wartość logiczną „prawda", albo wartość „fałsz"), więc wyrażenie takie może występować w roli operandu innego wyrażenia z operatorem logicznym. Stąd poprawność następującego wyrażenia:

```
eax < 0 && ebx <> eax   &&   !ecx
```

Operator iloczynu logicznego cechuje się łącznością lewostronną, co oznacza, że w kodzie wygenerowanym przez kompilator HLA ewaluacja (wartościowanie) wyrażenia następuje od strony lewej do prawej. Jeśli rejestr EAX zawiera wartość większą od zera, pozostałe operandy nie będą sprawdzane. Analogicznie, jeśli EAX jest mniejszy od zera, ale EBX jest równy EAX, to sprawdzanie logicznej wartości wyrażenia `!ecx` zostanie pominięte — całe wyrażenie od razu otrzyma wartość `false`.

Operator sumy logicznej w języku HLA reprezentowany jest znakami ||. Podobnie, jak operator iloczynu, operator sumy logicznej wymaga określenia dwóch operandów będących poprawnymi wyrażeniami logicznymi. Operator zwraca przy tym wartość logiczną „prawda", jeśli którykolwiek z operandów ma wartość logiczną „prawda". Również podobnie jak w przypadku operatora iloczynu stosowana jest metoda skróconej ewaluacji wyrażenia — jeśli już pierwszy operand ma wartość logiczną „prawda", operator natychmiast zwraca wartość „prawda", niezależnie od wartości drugiego operandu. Oto przykłady zastosowania operatora sumy logicznej:

```
@z || al = 10
al in 'a'..'z' || ebx
!boolVar || eax
```

Podobnie jak operator && operator logicznej sumy cechuje się łącznością lewostronną, więc w wyrażeniach z wieloma operatorami || ewaluacja przebiega od strony lewej do prawej. Na przykład:

```
eax < 0 || ebx <> eax    ||    !ecx
```

Jeśli powyższe wyrażenie zostałoby osadzone w wyrażeniu logicznym instrukcji `if`, to kod po klauzuli `then` zostałby wykonany tylko wtedy, kiedy rejestr EAX miałby wartość mniejszą od zera lub EBX nie byłby równy EAX bądź rejestr ECX zawierałby wartość zero. Jeśli już pierwszy test dałby wartość `true`, pozostałe warunki nie byłyby sprawdzane. Gdyby pierwszy warunek miał wartość `false`, ale drugi `true`, to etap szacowania wartości logicznej wyrażenia `!ecx` zostałby pominięty. Test zawartości rejestru ECX nastąpiłby jedynie w obliczu niespełnienie wcześniejszych dwóch wyrażeń.

Jeśli w jednym wyrażeniu logicznym występują operatory iloczynu i sumy logicznej, pierwszeństwo przed operatorem sumy ma operator iloczynu. Weźmy następujące wyrażenie:

```
eax < 0 || ebx <> eax && !ecx
```

W interpretacji HLA powyższemu wyrażeniu równoważne jest wyrażenie następujące:

```
eax < 0 || (ebx <> eax && !ecx)
```

Jeśli rejestr EAX będzie miał wartość mniejszą od zera, reszta wyrażenia nie będzie sprawdzana — całość wyrażenia otrzyma wartość `true`. Jeśli jednak rejestr EAX będzie

zawierał wartość większą od zera, to aby całość wyrażenia otrzymała wartość `true`, oba operandy operatora iloczynu musiałyby mieć wartość `true`.

Podwyrażenia wykorzystujące operatory iloczynu i sumy logicznej można grupować wedle własnego uznania, otaczając je nawiasami. Oto przykładowe wyrażenie:

```
(eax < 0 || ebx <> eax) && !ecx
```

Aby to wyrażenie otrzymało wartość logiczną „prawda", rejestr ECX musi zawierać wartość zero oraz albo wartość EAX musi być mniejsza od zera, albo rejestr EBX musi zawierać wartość różną od rejestru EAX. Szacowanie wyrażenia przebiega więc odmiennie niż w wersji bez nawiasów.

Operator negacji logicznej ma w języku HLA postać znaku wykrzyknika (`!`). Operandami operatora negacji mogą być jednak wyłącznie rejestry i zmienne w pamięci (np. `!eax < 0`) — operator negacji nie może być bezpośrednio aplikowany do złożonych wyrażeń logicznych. Aby wykonać logiczną negację wyrażenia logicznego, należy to wyrażenie ująć w nawiasy i cały nawias opatrzyć przedrostkiem w postaci wykrzyknika:

```
!( eax < 0)
```

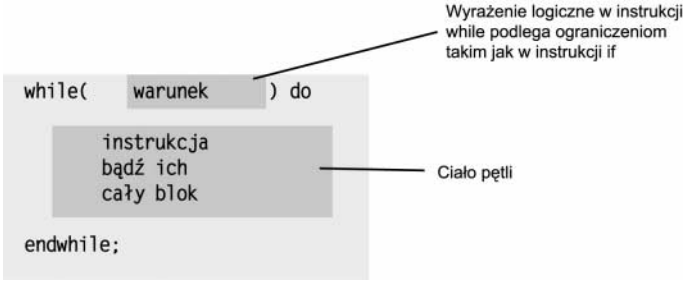
Powyższe wyrażenie otrzymuje wartość `true` wtedy, kiedy rejestr EAX ma wartość mniejszą od zera.

Operator logicznej negacji jest szczególnie użyteczny w odniesieniu do wyrażeń złożonych angażujących operatory logicznej sumy i logicznego iloczynu. W prostych wyrażeniach, jak powyższe, lepiej jest zazwyczaj bezpośrednio określić stan logiczny wyrażenia niż dodatkowo komplikować go operatorem negacji.

W języku HLA programista może też korzystać z operatorów bitowych `|` i `&`, ale te różnią się znacząco od operatorów logicznych `||` oraz `&&`. Szczegółowe omówienie operatorów bitowych i innych operatorów języka HLA znajduje się w dokumentacji kompilatora.

1.9.4. Instrukcja while

Składnię instrukcji `while` ilustruje rysunek 1.11.



Rysunek 1.11. *Składnia instrukcji while języka HLA*

Wykonanie instrukcji `while` przebiega następująco: najpierw szacowana jest wartość logiczna wyrażenia warunku pętli. Jeśli ma ono wartość „fałsz", sterowanie natychmiast przekazywane jest do kodu znajdującego się za klauzulą `endwhile`. W przypadku

wartości „prawda” wyrażenia warunku, procesor przystępuje do wykonania ciała pętli. Po wykonaniu wszystkich instrukcji ciała pętli następuje ponowne oszacowanie wartości logicznej wyrażenia warunkowego. Cała procedura powtarzana jest aż do momentu, w którym wyrażenie warunkowe pętli przyjmie wartość logiczną „fałsz”.

W pętli `while` sprawdzenie wartości warunku wykonania pętli następuje podobnie jak w językach wysokiego poziomu, czyli przed wykonaniem pierwszej instrukcji ciała pętli. Z tego względu istnieje prawdopodobieństwo, że instrukcje ciała pętli nie zostaną wykonane ani razu, jeśli wyrażenie warunku będzie od początku miało wartość `false`. Poza tym warto pamiętać, że przynajmniej jedna z instrukcji ciała pętli powinna modyfikować wartość wyrażenia logicznego — w przeciwnym przypadku będziemy mieli do czynienia z pętlą nieskończoną.

Oto przykład pętli `while` w języku HLA:

```
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;
```

1.9.5. Instrukcja for

W języku HLA pętla for przybiera następującą postać:

```
for( wyrażenie-inicjalizujące; warunek; instrukcja-licznika) do

    ciało-pętli

endfor;
```

Instrukcja pętli `for` odpowiada następującej pętli `while`:

```
wyrażenie-inicjalizujące;
while( warunek ) do

    ciało-pętli
    instrukcja-licznika;

endwhile;
```

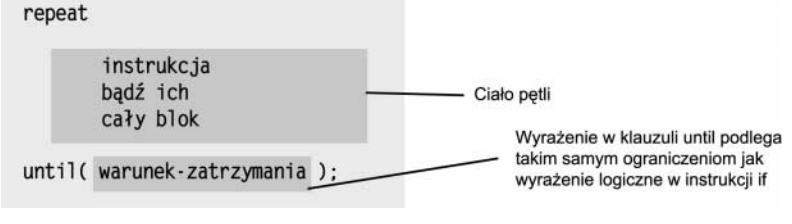
Wyrażenie inicjalizujące może być dowolną, pojedynczą instrukcją maszynową. Instrukcja ta zwykle inicjalizuje rejestr albo zmienną przechowywaną w pamięci, której wartość decyduje o kontynuowaniu pętli (tzw. **licznik pętli**). Warunek to wyrażenie logiczne języka HLA w formacie takim jak dla instrukcji `while`. Wartość tego wyrażenia decyduje o podjęciu kolejnej iteracji pętli i zwykle angażuje licznik pętli. Instrukcja licznika to typowo instrukcja zwiększająca licznik pętli; odpowiednikiem tej instrukcji jest instrukcja w pętli `while`, której wykonanie zmienia wartość logiczną warunku wykonania pętli. Zwykle instrukcją tą jest instrukcja języka HLA, na przykład `add`.

Oto przykład ilustrujący równoważność pętli `for` i `while`:

```
for( mov( 0, i ); i < 10; add( 1, i ) ) do

   stdout.put( "i=", i, nl );

endfor;

// Analogiczna pętla for:
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;
```

1.9.6. Instrukcja repeat

Instrukcja `repeat..until` języka HLA wykorzystuje składnię prezentowaną na rysunku 1.12. Programiści języków C, C++ oraz Java powinni od razu zauważyć, że instrukcja ta jest odpowiednikiem dostępnej w tych językach instrukcji `do..while`.



Rysunek 1.12. *Składnia instrukcji repeat..until języka HLA*

Test warunku zatrzymania pętli `repeat..until` odbywa się po zakończeniu wykonywania ciała pętli. Z tego względu instrukcja (bądź blok instrukcji) ciała pętli zostanie wykonana przynajmniej raz, niezależnie od wartości początkowej wyrażenia warunku zatrzymania pętli. Warunek ten jest badany po raz pierwszy dopiero po zakończeniu pierwszej iteracji; jeśli wyrażenie ma wartość `false`, pętla jest powtarzana[2]. Jeśli warunek otrzyma wartość `true`, pętla jest zatrzymywana i sterowanie przekazywane jest do instrukcji znajdujących się za klauzulą `until`.

Oto prosty przykład wykorzystania pętli `repeat..until`:

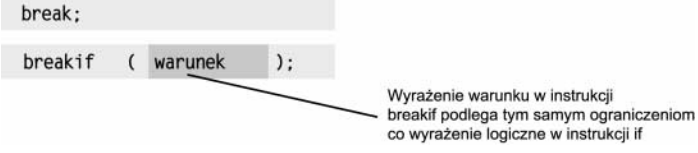
```
mov( 10, ecx );

repeat

    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

[2] Znaczenie warunku jest więc w języku HLA odwrotne niż warunku w analogicznej konstrukcji `do..while` znanej z języków C, C++ i Java.

Jeśli instrukcje ciała pętli mają być wykonane przynajmniej raz, niezależnie od wartości wyrażenia warunku pętli, wtedy w miejsce pętli `while` lepiej zastosować pętlę `repeat..until` — konstrukcja taka będzie efektywniejsza.

1.9.7. Instrukcje break oraz breakif

Instrukcje `break` i `breakif` służą do przedwczesnego przekazywania sterowania poza pętlę. Składnię obu instrukcji ilustruje rysunek 1.13.



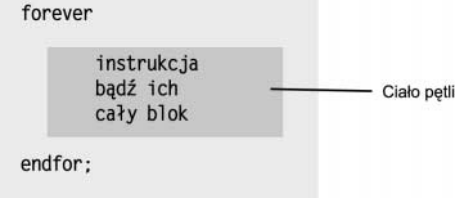
Rysunek 1.13. *Składnia instrukcji break oraz breakif języka HLA*

Instrukcja `break` powoduje bezwarunkowe przerwanie wykonywania pętli; instrukcja `breakif` uzależnia przekazanie sterowania poza pętlę od spełnienia warunku będącego wyrażeniem logicznym. Wyjście z pętli następuje wtedy wyłącznie w obliczu wartości `true` tego wyrażenia.

Zastosowanie instrukcji `break` oraz `breakif` nie pozwala na przekazanie sterowania poza pętle zagnieżdżone. W języku HLA służą do tego osobne instrukcje, jak `begin..end` oraz `exit` i `exitif`. Szczegółowe informacje na ten temat zawiera dokumentacja języka HLA. Dalej, język HLA udostępnia instrukcje `continue` oraz `continueif` pozwalające na przystąpienie do kolejnej iteracji pętli z pominięciem niewykonanych jeszcze instrukcji ciała pętli. Po szczegóły odsyłam Czytelników do stosownej dokumentacji.

1.9.8. Instrukcja forever

Składnię konstrukcji `forever..endfor` prezentuje rysunek 1.14.



Rysunek 1.14. *Składnia instrukcji forever języka HLA*

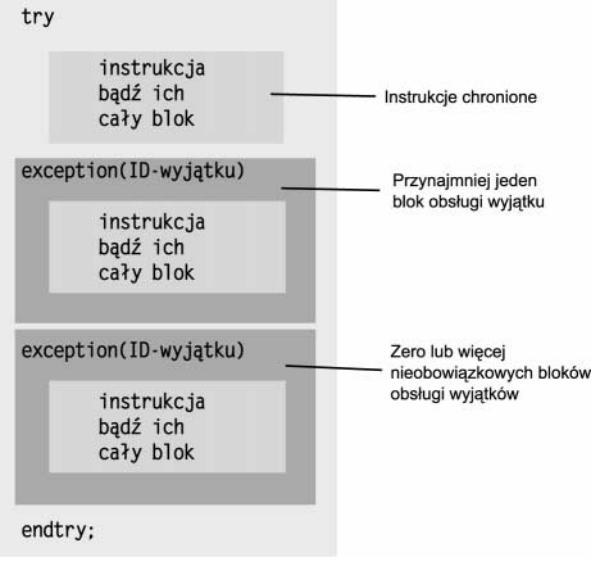
Instrukcja `forever` tworzy pętlę nieskończoną. Wewnątrz takiej pętli można jednak umieścić instrukcje `break` bądź `breakif` celem przerwania pętli w wybranym momencie. To chyba najczęstszy układ pętli `forever`. Ilustruje go następujący przykład:

```
forever

    stdout.put( "Wprowadz liczbe calkowita (mniejsza od 10):" );
    stdin.get( i );
    breakif( i < 10 );
    stdout.put( "Wprowadzona wartosc miala byc mniejsza od 10!", nl );

endfor;
```

1.9.9. Instrukcje try, exception oraz endtry

Instrukcje `try..exception..endtry` służą do implementacji bardzo przydatnych bloków **obsługi wyjątków**. Składnia instrukcji prezentowana jest na rysunku 1.15.

Rysunek 1.15. *Składnia instrukcji obsługi wyjątków*



Blok instrukcji `try..endtry` służy do ochrony wykonania instrukcji. Jeśli wykonanie instrukcji znajdujących się pomiędzy klauzulą `try` a pierwszą klauzulą `exception`, tworzących tzw. **blok chroniony**, przebiegnie bez niespodzianek, sterowanie przekazywane jest do pierwszej instrukcji za klauzulą `endtry`. Jeśli jednak w czasie wykonywania instrukcji bloku chronionego zdarzy się błąd (wyjątek), wykonanie programu jest przerywane, a sterowanie przekazywane jest do odpowiedniej klauzuli `exception` (skojarzonej z wyjątkiem, który wystąpił). Każdy wyjątek ma unikalny identyfikator w postaci liczby całkowitej. Niektóre z identyfikatorów wyjątków zadeklarowane zostały w pliku nagłówkowym *excepts.hhf* biblioteki standardowej języka HLA, ale programista może też definiować własne wyjątki. W momencie sprowokowania wyjątku system porównuje identyfikator zgłoszonego wyjątku z identyfikatorami określonymi w klauzulach `exception`. Jeśli wśród nich znajduje odpowiedni identyfikator, następuje wykonanie instrukcji określonych w ramach klauzuli `exception`. Po zakończeniu wykonywania tych instrukcji sterowanie przekazywane jest do pierwszej instrukcji za klauzulą `endtry`.

Jeśli wyjątek zostanie sprowokowany poza blokiem kodu chronionego (tzn. poza ramami instrukcji `try..endtry`) albo jeśli w ramach instrukcji `try..endtry` brak jest klauzuli skojarzonej ze zgłoszonym wyjątkiem, program jest awaryjnie zatrzymywany. Zatrzymaniu towarzyszy wyprowadzenie stosownego komunikatu na wyjście programu.

Instrukcje `try..endtry` można na przykład wykorzystać do ochrony programu przed skutkami wprowadzenia niepoprawnych danych:

```
repeat

    mov( false, GoodInteger );    // Uwaga: zmienna GoodInteger musi być typu boolean
    try

        stdout.put( "Wprowadz liczbe calkowita: " );
        stdin.get( i );
        mov( true, GoodInteger );

    exception( ex.ConversionError )

        stdout.put( "Niepoprawna wartosc, wprowadz jeszcze raz", nl );

    exception( ex.ValueOutOfRange )

        stdout.put( "Wartosc spoza zakresu, wprowadz jeszcze raz", nl );

    endtry;

until( GoodInteger );
```

Zastosowana w powyższym programie pętla `repeat` pozwala na wykonywanie kodu ciała pętli tak długo, jak długo użytkownik podaje niepoprawne wartości. Jeśli z powodu wprowadzenia niepoprawnej wartości jej przypisanie do zmiennej `GoodInteger` będzie niemożliwe (na przykład z racji umieszczenia we wprowadzonym niedozwolonych dla liczb znaków albo wprowadzenia zbyt wielkiej wartości), wykonany zostanie odpowiedni do rodzaju błędu blok instrukcji obsługi wyjątku. Obsługa wyjątków polega tu na wyświetleniu stosownego komunikatu i przekazaniu sterowania poza instrukcję `try..endtry`. Ponieważ z racji wyjątku nie udało się w ramach bloku chronionego ustawić wartości `true` dla zmiennej `GoodInteger`, pętla jest wykonywana ponownie. Jeśliby jedna w ramach bloku chronionego sprowokowany został inny niż obsługiwane wyjątek, program zakończyłby działanie po uprzednim wyprowadzeniu komunikatu o błędzie[3].

W tabeli 1.4 zebrane zostały wyjątki, zdefiniowane w pliku *excepts.hhf* w czasie przygotowywania wydania niniejszej książki. Aktualną listę zdefiniowanych w bibliotece standardowej wyjątków można znaleźć, zaglądając do pliku *excepts.hhf* dołączonego do posiadanej wersji oprogramowania kompilatora HLA.

Większość z tych wyjątków dotyczy sytuacji, których omawianie w niniejszym rozdziale byłoby przedwczesne. Zostały umieszczone w tabeli wyłącznie gwoli kompletności omówienia. Każdy z wyjątków opisany jest szczegółowo w dokumentacji *HLA Reference Manual* dokumentacji biblioteki standardowej języka HLA; ostateczną wykładnią jest zaś sam kod źródłowy biblioteki standardowej HLA. Najczęściej wykorzystywanymi wyjątkami są `ex.ConversionError`, `ex.ValueOutOfRange` oraz `ex.StringOverflow`.

Omówienie bloku kodu chronionego podjęte zostanie ponownie nieco później. Czytelnik powinien kontynuować edukację w zakresie podstawowym, zadowalając się na razie szczątkowymi informacjami o obsłudze wyjątków.

[3] Doświadczony programista zastanawia się zapewne, dlaczego w powyższym kodzie do sterowania przebiegiem pętli wykorzystywana jest zmienna typu `boolean` — przecież można by identyczny efekt osiągnąć, stosując instrukcję `breakif`. Zastosowana konstrukcja ma uzasadnienie, które zostanie przedstawione w dalszej części książki.

Tabela 1.4. *Wyjątki zdefiniowane w pliku nagłówkowym excepts.hhf*

Identyfikator wyjątku	Znaczenie
ex.StringOverflow	Próba zapisania zbyt długiego łańcucha
ex.StringIndexError	Próba odwołania do znaku o indeksie spoza łańcucha
ex.ValueOutOfRange	Zbyt wielka wartość dla danego typu zmiennej
ex.IllegalChar	Zaangażowanie w operacji znaku spoza dopuszczalnego zakresu (kod ASCI spoza zakresu od 0 do 127)
ex.ConversionError	Niedozwolone znaki w konwersji łańcucha do wartości numerycznej
ex.BadFileHandler	Odwołanie do pliku za pośrednictwem niepoprawnego uchwytu pliku
ex.FileOpenFailure	Niemożność otwarcia pliku (np. brak pliku o zadanej nazwie)
ex.FileCloseError	Niemożność zamknięcia pliku
ex.FileWriteError	Błąd przy zapisie danych do pliku
ex.FileReadError	Błąd przy odczycie danych z pliku
ex.DiskFullError	Próba zapisania danych do pliku przy pełnym dysku
ex.EndOfFile	Próba odczytania bajta spoza końca pliku
ex.MemoryAllocationFailure	Brak pamięci do zaspokojenia żądania alokacji
ex.AttemptToDerefNULL	Próba odwołania do pamięci za pośrednictwem wskaźnika o wartości NULL
ex.CannotFreeMemory	Błąd przy operacji zwalniania pamięci
ex.WidthTooBig	Zły format konwersji wartości liczbowej do łańcucha
ex.TooManyCmdLnParams	Wiersz wywołania programu zawiera zbyt dużą liczbę argumentów
ex.ArrayShapeViolation	Próba operowania na dwóch tablicach o różnych rozmiarach
ex.ArrayBounds	Próba odwołania do elementu spoza tablicy
ex.InvalidDate	Operacja na niepoprawnej wartości daty
ex.InvalidDateFormat	Błąd konwersji łańcucha do daty (łańcuch zawiera niedozwolone znaki)
ex.TimeOverflow	Przepełnienie w operacji manipulowania informacją o czasie
ex.AssertionFailed	Błąd instrukcji asercji
ex.ExecutedAbstract	Próba wywołania metody klasy abstrakcyjnej
ex.AccessViolation	Próba odwołania się do niedozwolonego adresu pamięci
ex.BreakPoint	Program dotarł do punktu zatrzymania (przerwanie 3.)
ex.SingleStep	Program wykonywany jest z ustawionym znacznikiem wykonania krokowego
ex.PrivInstr	Próba wykonania instrukcji charakterystycznej dla trybu nadzoru
ex.IllegalInstr	Próba wykonania niedozwolonej instrukcji maszynowe
ex.BoundInstr	Rejestr zawiera wartość spoza zakresu określonego w instrukcji `bound`
ex.IntoInstr	Wykonanie instrukcji `into` przy ustawionym znaczniku przepełnienia
ex.DivideError	Próba dzielenia przez zero bądź inny błąd dzielenia
ex.fDenormal	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
ex.fDivByZero	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)

Tabela 1.4. *Wyjątki zdefiniowane w pliku nagłówkowym excepts.hhf — ciąg dalszy*

Identyfikator wyjątku	Znaczenie
`ex.fInexactResult`	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
`ex.fInvalidOperation`	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
`ex.fOverflow`	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
`ex.fStackCheck`	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
`ex.fUnderflow`	Wyjątek operacji zmiennoprzecinkowej (patrz rozdział poświęcony obliczeniom arytmetycznym)
`ex.InvalidHandle`	System operacyjny odrzucił niepoprawny uchwyt
`ex.StackOverflow`	System operacyjny zgłosił przepełnienie stosu

1.10. Biblioteka standardowa języka HLA — wprowadzenie

Język HLA prezentuje się na tle tradycyjnego języka asemblerowego korzystnie dzięki dwóm aspektom. Pierwszym jest składnia języka HLA zapożyczająca elementy znane z języków programowania wysokiego poziomu (jak struktury kontroli wykonania kodu czy deklaracje zmiennych). Dostępność tych elementów pozwala na efektywne wykorzystanie umiejętności przeniesionych z programowania wysokopoziomowego i tym samym znacznie szybszą naukę języka asemblerowego. Drugim elementem stanowiącym o przewadze HLA nad tradycyjnym asemblerem jest dostępność biblioteki HLA Standard Library. Biblioteka standardowa języka HLA zawiera wiele prostych w użyciu procedur języka asemblerowego realizujących typowe zadania programistyczne — ich dostępność eliminuje konieczność samodzielnego pisania wszystkich, nawet tych najbardziej typowych, elementów programu. Dostępność biblioteki standardowej znosi jedną z podstawowych trudności, w obliczu których stają programiści chcący poznać język asemblerowy — trudność tkwiącą w konieczności samodzielnej implementacji złożonych niekiedy procedur wejścia-wyjścia, niezbędnych w nawet najprostszych programach. Brak standaryzowanej biblioteki wywołań powodowałby, że każdy początkujący programista poświęcałby mnóstwo czasu na zaimplementowanie tak podstawowych operacji jak wyprowadzanie napisów. Jej dostępność oznacza więc, że adept języka asemblerowego może skoncentrować się na nauce samego języka, nie poświęcając się od samego początku zgłębianiu tajników podsystemu wejścia-wyjścia danego systemu operacyjnego.

Dostępność rozmaitych procedur bibliotecznych to nie jedyna zaleta języka HLA. W końcu biblioteki kodu asemblerowego to nie nowy wynalazek[4]. Jednak biblioteka standardowa

[4] Popularna jest na przykład biblioteka UCR Standard Library dla programistów języka asemblerowego procesorów 80x86.

języka HLA to nie tylko sam kod procedur, ale również wysokopoziomowy interfejs wywołań tych procedur. W rzeczy samej język HLA powstał głównie z myślą o możliwości tworzenia zestawów procedur bibliotecznych wysokiego poziomu. Ów wysokopoziomowy interfejs wywołań, w połączeniu z wysokopoziomową naturą wielu procedur, daje programiście niespodziewanie wielkie i równocześnie łatwo dostępne możliwości.

Biblioteka standardowa języka HLA składa się z szeregu modułów przypisanych do kilku kategorii. Część z dostępnych modułów zawiera tabela 1.5[5].

Tabela 1.5. *Wybrane moduły biblioteki standardowej HLA*

Nazwa modułu	Funkcje
args	Procedury przetwarzania argumentów wiersza polecenia wraz z procedurami pomocniczymi
conv	Procedury konwersji łańcuchów do wartości różnych typów i konwersji odwrotnych
cset	Procedury operujące na zbiorach znaków
DateTime	Procedury manipulowania datą i czasem
excepts	Procedury obsługi wyjątków
fileio	Procedury plikowych operacji wejścia-wyjścia
hla	Specjalne stałe języka HLA i inne definicje
hll	Implementacja instrukcji znanych z języków wysokopoziomowych
Linux	Wywołania systemowe jądra systemu Linux (tylko w wersji HLA dla systemu Linux)
math	Procedury implementujące obliczenia arytmetyczne o zwiększonej precyzji, funkcje trygonometrycznych i inne funkcje matematyczne
memory	Procedury przydziału i zwalniania pamięci wraz z procedurami pomocniczymi
misctypes	Definicje rozmaitych typów danych i operacji na tych typach
patterns	Biblioteka procedur dopasowywania wzorców
rand	Implementacja generatora liczb pseudolosowych i procedur pomocniczych
stdin	Procedury obsługi wejścia
stdout	Procedury obsługi wyjścia
stdlib	Specjalny plik nagłówkowy grupujący wszystkie moduły biblioteki standardowej języka HLA
strings	Biblioteka procedur manipulowania łańcuchami znakowymi
tables	Biblioteka procedur obsługi tablic asocjacyjnych
win32	Definicje stałych wykorzystywanych w wywołaniach systemu Windows (tylko w wersji HLA dla środowiska Win32)
x86	Stałe i definicje charakterystyczne dla procesora 80x86

[5] Z racji ciągłego rozwoju biblioteki standardowej języka HLA niniejsza lista nie jest zapewne kompletna. Aktualnej listy modułów należy szukać w dokumentacji języka HLA.

Część z tych modułów doczeka się szczegółowego omówienia w dalszej części książki. Tutaj skupimy się na najważniejszych — z punktu widzenia początkującego programisty — procedurach biblioteki standardowej HLA, zebranych w module `stdio`.

1.10.1. Stałe predefiniowane w module stdio

Omówienie modułu `stdio` biblioteki standardowej należałoby rozpocząć od przedstawienia najczęściej wykorzystywanych stałych. Na przykład w następującym kodzie:

```
stdout.put( "Ahoj, przygodo!", nl );
```

pojawia się stała `nl` reprezentująca sekwencję znaków przejścia do nowego wiersza. Identyfikator `nl` nie jest w kontekście programu HLA żadnym słowem zarezerwowanym języka HLA; jego zastosowanie nie ogranicza się też do wywołania `stdout.put`. Otóż `nl` to jedynie stała, której wartością jest literał łańcuchowy odpowiadający sekwencji znaków nowego wiersza (w systemie Windows sekwencja ta obejmuje znaki powrotu karetki i wysuwu wiersza; w Linuksie to pojedynczy znak wysuwu wiersza).

Oprócz stałej `nl` implementacja podsystemu wejścia-wyjścia w bibliotece standardowej HLA zawiera definicje kilku innych przydatnych stałych znakowych. Zostały one wymienione w tabeli 1.6.

Tabela 1.6. *Stałe znakowe definiowane w module wejścia-wyjścia biblioteki standardowej HLA*

Stała	Definicja
`stdio.bell`	Znak dzwonka; wydrukowanie tego znaku objawia się wygenerowaniem dźwięku przez głośniczek systemowy
`stdio.bs`	Znak cofania kursora
`stdio.tab`	Znak tabulacji
`stdio.lf`	Znak wysuwu wiersza
`stdio.cr`	Znak powrotu karetki

Z wyjątkiem `nl` wszystkie zaprezentowane tu stałe zdefiniowane są w **przestrzeni nazw** `stdio` (stąd wymóg poprzedzania nazw stałych przedrostkiem `stdio`)[6]. Osadzenie owych stałych w przestrzeni nazw `stdio` pozwoliło na uniknięcie ewentualnych konfliktów z nazwami zmiennych definiowanymi przez programistę. Nazwa `nl`, jako używana wyjątkowo często, nie została osadzona w przestrzeni nazw `stdio` — konieczność każdorazowego wpisywania `stdio.nl` byłaby bardzo uciążliwa.

1.10.2. Standardowe wejście i wyjście programu

Wiele procedur wejścia-wyjścia biblioteki HLA jest prefiksowanych nazwą `stdin` bądź `stdout`. Technicznie oznacza to, że nazwy tych procedur zdefiniowane zostały wewnątrz odpowiedniej przestrzeni nazw. W praktyce zastosowanie przedrostka stanowi również sugestię co do pochodzenia danych (dane pobierane ze standardowego urządzenia

[6] Przestrzenie nazw omówione zostaną w jednym z kolejnych rozdziałów.

wejściowego, stdin) bądź ich przeznaczenia (dane kierowane do standardowego urządzenia wyjściowego, stdout). Najczęściej urządzeniem wejściowym jest klawiatura konsoli systemowej. Urządzenie wyjściowe to zwykle ekran konsoli. W ogólności więc instrukcje z przedrostkiem stdin bądź stdout realizują odczyty i zapisy danych z i do urządzenia konsoli.

Przy uruchamianiu programu z poziomu wiersza poleceń (zwanego w Linuksie powłoką) można dokonać **przekierowania** strumieni danych wejściowych i wyjściowych do urządzeń innych niż domyślne. Na przykład argument wywołania programu w postaci >plik_wy powoduje przekierowanie standardowego wyjścia programu do pliku *plik_wy*. Z kolei argument wywołania postaci <plik_we powoduje przypisanie standardowego wejścia programu do pliku źródłowego *plik_we*. Oto przykłady przekierowania standardowego wejścia i wyjścia programu testpgm uruchamianego z poziomu wiersza poleceń[7]:

```
testpgm <input.data
testpgm >output.txt
testpgm <in.txt >out.txt
```

1.10.3. Procedura stdout.newln

Procedura stdout.newln powoduje zapisanie do urządzenia standardowego wyjścia sekwencji nowego wiersza. Wywołanie owej procedury jest równoważne wywołaniu stdout.put(nl);. Wywołanie stdout.newln jest jednak niekiedy nieco wygodniejsze bądź bardziej czytelne. Oto przykład wywołania procedury:

```
stdout.newln();
```

1.10.4. Procedury stdout.putiN

Procedury biblioteczne stdout.puti8, stdout.puti16 oraz stdout.puti32 wyprowadzają na standardowe wyjście programu pojedynczą wartość, odpowiednio: 8-bitową, 16-bitową i 32-bitową, interpretowaną jako liczba całkowita ze znakiem. Argumentem wywołania może być stała, rejestr bądź zmienna w pamięci — rozmiar argumentu musi jednak odpowiadać rozmiarowi parametru formalnego procedury.

Omawiane procedury wyprowadzają przekazaną wartość na standardowe wyjście programu. Wyprowadzany napis reprezentujący wartość konstruowany jest z użyciem minimalnej liczby znaków pozwalających na dokładne odwzorowanie wartości. Jeśli przekazana argumentem wywołania procedury wartość jest ujemna, napis zostanie poprzedzony znakiem minusa. Oto kilka przykładów wywołań procedur stdout.puti*N*:

```
stdout.puti8( 123 );
stdout.puti16( dx );
stdout.puti32( i32Var );
```

[7] Uwaga dla użytkowników systemu Linux: w zależności od wartości zmiennej środowiskowej PATH nazwę programu trzeba być może poprzedzić znakami „./”, np. ./testpgm <input.data.

1.10.5. Procedury stdout.putiNSize

Procedury `stdout.puti8Size`, `stdout.puti16Size` oraz `stdout.puti32Size` zapisują do standardowego urządzenia wyjściowego wartości interpretowane jako liczby całkowite ze znakiem — a więc działają podobnie jak procedury `stdout.putiN`. Tyle że procedury `stdout.putiNSize` pozwalają lepiej kontrolować format wyprowadzanych napisów — programista może odpowiednim argumentem wywołania procedury określić minimalną szerokość wyprowadzanego napisu. W przypadku kiedy wyprowadzana wartość po konwersji będzie zajmowała mniejszą niż podano liczbę znaków, brakujące znaki uzupełniane są znakami wypełnienia — znak wypełnienia również określa się argumentem wywołania procedury. Składnia wywołania procedur `stdout.putiNSize` jest następująca:

```
stdout.puti8Size( wartość-8-bit, szerokość, wypełnienie );
stdout.puti16Size( wartość-16-bit, szerokość, wypełnienie );
stdout.puti32Size( wartość-32-bit, szerokość, wypełnienie );
```

W miejsce parametru *wartość-N-bit* można przekazać stałą, nazwę rejestru albo nazwę zmiennej w pamięci — podany argument musi jednak odpowiadać rozmiarem rozmiarowi parametru. Parametr *szerokość* może przyjmować argumenty w postaci stałych całkowitych z zakresu od –256 do 256. Argumentem tego parametru może być również zawartość rejestru bądź wartość zmiennej przechowywanej w pamięci. Argumentem parametru *wypełnienie* powinien być pojedynczy znak.

Podobnie jak procedury `stdout.putiN` procedury `stdout.putiNSize` wyprowadzają na urządzenie standardowego wyjścia napis reprezentujący przekazaną w wywołaniu wartość liczbową interpretowaną jako liczbę całkowitą ze znakiem. W tej wersji programista może jednak określić szerokość napisu — określa ją jako minimalną liczbę znaków, jaką powinien zajmować napis. Jeśli wyświetlana wartość zajmować będzie więcej znaków (na przykład przy próbie wyświetlenia wartości 1234 w polu o szerokości 2), to wyprowadzony napis zostanie stosownie poszerzony. Jednak w przypadku, gdy liczba znaków w napisie jest mniejsza od założonego minimum, brakujące znaki są w kodzie procedury uzupełniane znakami wypełnienia. Kiedy wyprowadzana wartość argumentu *szerokość* jest liczbą ujemną, napis zostanie wyrównany w ramach wyprowadzanego pola do lewej; wartość dodatnia powoduje wyrównywanie napisów do prawej.

Jeśli wartość bezwzględna parametru *szerokość* jest większa niż minimalna liczba znaków potrzebnych do dokładnego reprezentowania wyprowadzanej wartości, procedury `stdout.putiNSize` uzupełniają napis znakami wypełnienia, dodając je z lewej bądź z prawej strony napisu (przed bądź za znakową reprezentacją liczby). Znak wypełnienia definiowany jest argumentem *wypełnienie*. W zdecydowanej większości przypadków znakiem tym jest znak spacji, jednak w sytuacjach szczególnych konieczne może okazać się wypełnienie innym znakiem. Należy tylko pamiętać, że argument parametru *wypełnienie* powinien być wartością znakową. W języku HLA stałe znakowe ujmowane są w znaki pojedynczego cudzysłowu. W miejsce tego parametru można też wskazać ośmiobitowy rejestr.

Listing 1.4 prezentuje prosty program w języku HLA demonstrujący sposób wykorzystania procedury `stdout.puti32Size` w celu wyświetlenia listy wartości w formie tabelarycznej.

Listing 1.4. *Wyprowadzenie kolumn wartości liczbowych — zastosowanie procedury stdout.puti32Size*

```
program NumsInColumns;

#include( "stdlib.hhf" )

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.puti32Size( i32, 5, ' ' );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.newln();

end NumsinColumns;
```

1.10.6. Procedura stdout.put

Procedura `stdout.put`[8] to najelastyczniejsza procedura wyprowadzania napisów na wyjście programu dostępna w module obsługi standardowego wyjścia. Procedura ta łączy funkcje wszystkich pozostałych procedur wyjścia standardowego, udostępniając programiście elegancki i efektywny mechanizm wyprowadzania danych.

W najbardziej ogólnej postaci składnia wywołania procedury `stdout.put` prezentuje się następująco:

```
stdout.put( lista-wyprowadzanych-wartości );
```

Lista argumentów wywołania procedury `stdout.put` może zostać konstruowana ze stałych, rejestrów i zmiennych; kolejne argumenty oddziela się przecinkami. Procedura wyprowadza na standardowe wyjście wartości wszystkich kolejnych argumentów wywołania. Jako że procedura ta była już wielokrotnie prezentowana w przykładach, Czytelnik orientuje się, przynajmniej w zakresie podstawowym, co do sposobu jej wywoływania.

[8] Tak naprawdę `stdout.put` to makrodefinicja, a nie procedura. Rozróżnienie pomiędzy makrodefinicją a procedurą wykracza jednak poza zakres niniejszego rozdziału; zostanie ono uwzględnione w dalszej części książki.

Warto jedynie podkreślić, że niniejsza procedura udostępnia programiście szereg właściwości, które nie były dotąd prezentowane w przykładach. W szczególności każdy z argumentów wywołania może być zadany w jednej z dwóch postaci:

```
wartość
wartość:szerokość
```

Wartość może być dowolną stałą, rejestrem albo zmienną przechowywaną w pamięci. W niniejszym rozdziale prezentowane były argumenty w postaci literałów łańcuchowych i zmiennych w pamięci. Jak dotychczas wszystkie argumenty wywołania odpowiadały pierwszej możliwej postaci argumentu. W postaci drugiej programista może po dwukropku określić minimalną szerokość napisu reprezentującego wartość — szerokość ta interpretowana jest podobnie jak w procedurach `stdout.putiNSize`[9]. Przykładowy program z listingu 1.5 generuje wydruk podobny do tego tworzonego programem `NumsInColumns`, tyle że tutaj miejsce procedury `stdout.putiNSize` zajęła procedura `stdout.put`.

Listing 1.5. *Demonstracja sposobu określenia szerokości napisów w argumentach procedury stdout.put*

```
program NumsInColumns2;

#include( "stdlib.hhf" )

var
    i32:     int32;
    ColCnt:  int8;

begin NumsInColumns2;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.put( i32:5 );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.newln();

end NumsinColumns;
```

Procedura `stdout.put` obsługuje jednak więcej atrybutów. Będą one omawiane w kolejnych rozdziałach, w miarę potrzeby ich wprowadzania do programów przykładowych.

[9] W odróżnieniu od procedur *stdout.putiNSize* nie da się tu określić dowolnego znaku wypełnienia — znakiem tym jest domyślnie znak spacji. W obliczu konieczności zastosowania innego wypełnienia należy skorzystać z procedur *stdout.putiNSize*.

1.10.7. Procedura stdin.getc

Procedura `stdin.getc` wczytuje pojedynczy znak z bufora urządzenia standardowego wejścia[10]. Znak ten umieszczany jest po wyjściu z procedury w rejestrze AL. Zastosowanie procedury ilustruje program z listingu 1.6.

Listing 1.6. *Demonstracja działania procedury stdin.getc*

```
program charInput;

#include( "stdlib.hff" )

var
    counter: int32;

begin charInput;

    // Poniższa pętla wykonywana jest dopóty,
    // dopóki użytkownik potwierdza wykonanie kolejnej iteracji.

    repeat

        // Wyprowadź 14 wartości.

        mov( 14, counter );
        while( counter > 0 ) do

            stdout.put( counter:3 );
            sub( 1, counter );

        endwhile;

        // Zaczekaj, aż użytkownik naciśnie 't' albo 'n'.

        stdout.put( nl, nl, "Wykonac kolejna iteracje (t/n)?:" );
        forever

            stdin.readLn();
            stdin.getc();
            breakif( al = 'n' );
            breakif( al = 't' );
            stdout.put( "Prosze wprowadzic albo 't', albo 'n':" );

        endfor;
        stdout.newln();

    until( al = 'n' );

end charInput;
```

[10] *Bufor* to w tym kontekście ładniejsza nazwa tablicy.

W powyższym programie wywołanie procedury `stdin.readLn` służy do wymuszenia przejścia do nowego wiersza strumienia wejściowego. Szczegółowy opis tej procedury znajduje się nieco dalej, w niniejszym rozdziale.

1.10.8. Procedury stdin.getiN

Procedury `stdin.geti8`, `stdin.geti16` oraz `stdin.geti32` realizują operację wczytania z urządzenia standardowego wejścia odpowiednio: 8-bitowej, 16-bitowej i 32-bitowej wartości liczbowej całkowitej. Wczytane wartości umieszczane są w rejestrach AL, AX bądź EAX. Niniejsze procedury stanowią standardowy sposób wczytywania liczb całkowitych ze znakiem do programów języka HLA.

Podobnie jak procedura `stdin.getc`, procedury `stdin.getiN` wczytują ze standardowego wejścia sekwencję znaków. W sekwencji tej ignorowane są wszelkie początkowe znaki odstępów (spacje, znaki tabulacji i tak dalej); reszta sekwencji (zawierająca cyfry poprzedzone opcjonalnym znakiem minusa) konwertowana jest do postaci liczbowej. W przypadku, kiedy sekwencja wejściowa zawiera znaki inne niż dozwolone lub wprowadzona wartość nie mieści się w zakresie liczbowym charakterystycznym dla danej procedury, zgłaszany jest wyjątek (można go przechwycić, umieszczając wywołanie procedury w bloku kodu chronionego w instrukcji `try..endtry`). Procedura `stdin.geti8` obsługuje liczby z zakresu od –128 do 127, `stdin.geti16` — od –32 768 do 32 767; wartości wczytywane do programu za pośrednictwem procedury `stdin.geti32` muszą zaś mieścić się w przedziale od –2 147 483 648 do 2 147 483 647.

Sposób wykorzystania owych procedur ilustruje listing 1.7.

Listing 1.7. *Przykład wykorzystania wywołań stdin.getiN*

```
program intInput;

#include( "stdlib.hhf" )

var:
    i8:     int8;
    i16:    int16;
    i32:    int32;

begin intInput;

    // Pobierz od użytkownika kilka liczb całkowitych różnej wielkości.

    stdout.put( "Wprowadz niewielka liczbe calkowita (z zakresu -128..127): " );
    stdin.geti8();
    mov( al, i8 );

    stdout.put( "Wprowadz wieksza liczbe calkowita (z zakresu -32 768..32 767): "
);
    stdin.geti16();
    mov( ax, i16 );

    stdout.put( "Wprowadz liczbe calkowita (z zakresu od -2 do 2 miliardow): " );
    stdin.geti32();
```

```
    mov( eax, i32 );

    // Wyświetl podane wartości.

    stdout.put
    (
        nl,
        "Oto wprowadzone liczby:", nl, nl,
        "8-bitowa liczba calkowita:  ", 18:12, nl,
        "16-bitowa liczba calkowita: ", 116:12, nl,
        "32-bitowa liczba calkowita: ", 132:12, nl
    );

end intInput;
```

Warto skompilować i uruchomić powyższy program i sprawdzić, co się stanie, jeśli wprowadzane liczby będą wykraczały poza dozwolony zakres albo ciągi wartości zawierać będą niedozwolone znaki.

1.10.9. Procedury stdin.readLn i stdin.flushInput

Moment wywołania procedury `stdin.getc` czy `stdin.geti32` nie musi być tożsamy z momentem wprowadzenia danych przez użytkownika. Biblioteka standardowa języka HLA buforuje dane wprowadzane na standardowe wejście, wczytując jednorazowo cały wiersz tekstu wprowadzanego na to wejście. Wywołanie procedury wejścia może więc zostać zrealizowane za pośrednictwem operacji odczytu danych z bufora wejściowego (o ile ten nie jest pusty). Choć buforowanie w ogólności poprawia efektywność manipulowania danymi wejściowymi, niekiedy wprowadza zamieszanie, jak w poniższym przykładzie:

```
stdout.put( "Wprowadz niewielka liczbe calkowita (z zakresu -128..127): " );
stdin.geti8();
mov( al, i8 );

stdout.put( "Wprowadz wieksza liczbe calkowita (z zakresu -32 768..32 767): " );
stdin.geti16();
mov( ax, i16 );
```

Programista ma nadzieję, że wykonanie powyższego kodu da następujący efekt: program wyświetli monit o podanie liczby, zaczeka na wprowadzenie jej przez użytkownika, wyświetli kolejny monit i również zaczeka na wprowadzenie liczby. W rzeczywistości program może mieć nieco inny przebieg. Jeśli, na przykład, powyższy kod zostanie uruchomiony, a użytkownik w odpowiedzi na pierwszy monit wprowadzi łańcuch „123 456" program nie będzie już po wyświetleniu drugiego monitu oczekiwał na wprowadzenie danych — procedura `stdin.geti16` odczyta po prostu drugi z wprowadzonych do bufora wejściowego łańcuchów (456).

Procedury wejścia oczekują na wprowadzenie tekstu przez użytkownika jedynie wtedy, kiedy bufor wejściowy jest pusty. Dopóki zawiera on jakiekolwiek znaki, procedury wejścia ograniczają się do odczytu z bufora. Można to wykorzystać, konstruując następujący kod:

```
stdout.put( "Wprowadz dwie liczby calkowite: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

Tutaj zezwala się wprost użytkownikowi na jednorazowe (w ramach pojedynczego wiersza) wprowadzenie dwóch liczb. Liczby te powinny zostać oddzielone jednym bądź kilkoma znakami odstępu. Pozwala to na zaoszczędzenie miejsca na ekranie. Tutaj buforowanie danych wejściowych okazało się korzystne. Kiedy indziej jednak może być odwrotnie.

Na szczęście biblioteka standardowa języka HLA przewiduje dwie procedury sterujące działaniem bufora standardowego urządzenia wejściowego. Są to procedury `stdin.readLn` oraz `stdin.flushInput`. Pierwsza z nich usuwa z bufora wszystko, co zostało w nim umieszczone wcześniej, i tym samym wymusza wprowadzenie przez użytkownika nowego wiersza danych. Druga po prostu usuwa całą zawartość bufora. Oczyszczenie bufora powoduje, że przy następnym wywołaniu dowolnej z procedur wejścia, użytkownik będzie musiał wprowadzić nowy wiersz danych. Procedurę `stdin.readLn` wywołuje się zwykle bezpośrednio przez wywołaniem dowolnej procedury wejścia. Procedura `stdin.flushInput` wywoływana jest zaś typowo bezpośrednio po odczytaniu danych z bufora wybraną procedurą wejścia.

Jeśli w programie wykorzystywana jest procedura `stdin.readLn` i okaże się, że konieczne jest dwukrotne wprowadzenie danych, to należy zastanowić się nad wykorzystaniem w jej miejsce procedury `stdin.flushInput`. W ogólności ta ostatnia procedura, pozwalająca na opróżnienie bufora przez kolejną operacją wejścia, jest wykorzystywana znacznie częściej. Konieczność zastosowania procedury `stdin.readLn` jest bardzo rzadka — należy ją stosować wyłącznie tam, gdzie konieczne jest wymuszenie na użytkowniku wprowadzenia nowego, aktualnego wiersza danych.

1.10.10. Procedura stdin.get

Procedura `stdin.get` łączy w sobie przedstawione wcześniej procedury wejścia, udostępniając ich funkcje za pośrednictwem pojedynczego wywołania. Procedura `stdin.get` jest przy tym nieco prostsza w użyciu niż `stdout.put`, ponieważ jedynymi argumentami wywołania tej pierwszej są nazwy zmiennych (lub rejestrów), w których mają zostać umieszczone wczytane wartości.

Przyjrzyjmy się ponownie przykładowi z poprzedniego punktu:

```
stdout.put( "Wprowadz dwie liczby calkowite: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

Równoważny przykład wykorzystujący procedurę `stdin.get` wyglądałby następująco:

```
stdout.put( "Wprowadz dwie liczby calkowite: " );
stdin.get( intval, AnotherIntVal );
```

Jak widać, zastosowanie procedury `stdin.get` może program skrócić i zwiększyć jego czytelność.

Należy przy tym pamiętać, że wywołanie procedury `stdin.get` powoduje umieszczenie wczytanych wartości od razu we wskazanych zmiennych; żadne z wczytanych wartości nie pojawią się w rejestrach, chyba że rejestry te wystąpią na liście argumentów wywołania. Argumentami wywołania mogą być zarówno nazwy zmiennych, jak i nazwy rejestrów.

1.11. Jeszcze o ochronie wykonania kodu w bloku try..endtry

Jak zapewne Czytelnik pamięta, instrukcje `try..endtry` otaczają blok kodu, którego wykonanie może potencjalnie sprowokować wyjątki zakłócające działanie programu. Wyjątki mogą mieć trzy źródła: mogą być zgłaszane sprzętowo (np. w przypadku dzielenia przez zero), generowane przez system operacyjny albo podnoszone wykonaniem odpowiedniej instrukcji języka HLA. Programista może przechwytywać i obsługiwać sytuacje wyjątkowe, umieszczając stosowny kod w ramach klauzuli `exception`. Typowy przykład zastosowania instrukcji `try..endtry` ilustruje listing 1.8.

Listing 1.8. *Przykład zastosowania bloku try..endtry*

```
program testBadInput;
#include( "stdlib.hhf" );

static:
    u:      uns16;

begin testBadInput;

    try

        stdout.put( "Wprowadz liczbe calkowita bez znaku: " );
        stdin.get( u );
        stdout.put( "Wprowadzono: ", u, nl );

    exception( ex.ConversionError )

        stdout.put( "Wprowadzony ciag zawiera niedozwolone znaki" nl );

    exception( ex.ValueOutOfRange )

        stdout.put( "Wprowadzona liczba jest zbyt duza" nl );

    endtry;

end testBadInput;
```

W języku HLA instrukcje znajdujące się za słowem `try`, a przed pierwszą klauzulą `exception` noszą miano instrukcji **chronionych**. Jeśli podczas wykonywania takich instrukcji zgłoszony zostanie wyjątek, sterowanie zostanie przekazane do pierwszej klauzuli `exception` skojarzonej z danym wyjątkiem, przeszukując kolejne klauzule pod kątem zgodności identyfikatora wyjątku zadeklarowanego ze zgłoszonym[11]. Identyfikator wyjątku to po prostu 32-bitowa liczba. Stąd również każda wartość umieszczana w nawiasach klauzuli `exception` powinna być wartością 32-bitową. Predefiniowane wartości wyjątków języka HLA wymienione są w pliku *excepts.hhf*. I choć byłoby to niewątpliwie rażącym naruszeniem przyjętego powszechnie stylu programowania, w klauzulach `exception` dozwolone jest podawanie wprost wartości numerycznych wyjątków bez korzystania z predefiniowanych nazw stałych.

1.11.1. Zagnieżdżone bloki try..endtry

Jeśli w wyniku przeszukania klauzul `exception` nie nastąpi dopasowanie identyfikatora wyjątku, będzie miało miejsce przeszukanie klauzul nadrzędnego bloku `try..endtry`, w którym blok bieżący jest **zagnieżdżony dynamicznie**. Spójrzmy na przykład z listingu 1.9.

Listing 1.9. *Zagnieżdżanie instrukcji try..endtry*

```
program testBadInput2;
#include( "stdlib.hhf" );

static:
    u:      uns16;

begin testBadInput2;

    try

        try

            stdout.put( "Wprowadz liczbe calkowita bez znaku: " );
            stdin.get( u );
            stdout.put( "Wprowadzono: ", u, nl );

        exception( ex.ConversionError )

            stdout.put( "Wprowadzony ciag zawiera niedozwolone znaki" nl );

        endtry;

        stdout.put( "Blad nie wynika z przepelnienia zakresu" nl );

    exception( ex.ValueOutOfRange )
```

[11] W programie HLA identyfikator ten umieszczany jest w rejestrze EAX. Stąd po przekazaniu sterowania do odpowiedniej klauzuli `exception` można odwoływać się do identyfikatora wyjątku, odwołując się do rejestru EAX.

```
            stdout.put( "Wprowadzona liczba jest zbyt duza" nl );

        endtry;

    end testBadInput2;
```

Na listingu 1.9 widać zagnieżdżone dwa bloki instrukcji `try..endtry`. Jeśli w ramach wykonania instrukcji `stdin.get` użytkownik wprowadzi wartość większą od 4 miliardów z kawałkiem, procedura `stdin.get` podniesie wyjątek `ex.ValueOutOfRange`. Kiedy wyjątek ten zostanie przekazany do systemu wykonawczego HLA, nastąpi przeszukanie klauzul `exception` bieżącego bloku `try..endtry` (tego, w którym nastąpiło zgłoszenie wyjątku; w prezentowanym przykładzie jest to blok zagnieżdżony). Jeśli nie uda się dopasować identyfikatora wyjątku `ex.ValueOutOfRange` w bieżącym bloku `try..endtry`, system wykonawczy HLA sprawdzi, czy bieżący blok `try..endtry` nie został zagnieżdżony w innym bloku `try..endtry` (co akurat ma miejsce na listingu 1.9). W takim przypadku system wykonawczy HLA przeszuka również klauzule `exception` bloku zewnętrznego (nadrzędnego). Na listingu 1.9 ów blok zawiera klauzulę z odpowiednią wartością identyfikatora, więc sterowanie przekazywane jest do pierwszej instrukcji umieszczonej za klauzulą `exception(ex.ValueOutOfRange)`.

Po opuszczeniu bloku `try..endtry` system wykonawczy traktuje ów blok jako nieaktywny i jeśli teraz nastąpi zgłoszenie wyjątku, system wykonawczy HLA nie będzie przeszukiwał klauzul `exception` tego bloku[12]. Pozwala to na różnicowanie procedur obsługi wyjątków w różnych miejscach programu.

Jeśli ten sam wyjątek obsługują dwa bloki `try..endtry`, a jeden z tych bloków jest zagnieżdżony w bloku chronionym pierwszego, to w momencie zgłoszenia wyjątku podczas wykonywania wewnętrznego bloku chronionego system wykonawczy HLA przekaże sterowanie do kodu obsługi wyjątku w wewnętrznym bloku `try..endtry`. Kod obsługi tego samego wyjątku w zewnętrznym bloku jest ignorowany — HLA nie przekazuje sterowania do tego kodu.

W przykładzie z listingu 1.9 druga instrukcja `try..endtry` została statycznie zagnieżdżona w pierwszym bloku `try..endtry`[13]. Wcześniej wspomniano, że jeśli bieżący blok `try..endtry` nie zawiera kodu obsługi zgłoszonego wyjątku, następuje przeszukanie klauzul `exception` kolejnych bloków `try..endtry`, w których blok bieżący został zagnieżdżony dynamicznie. W przypadku zagnieżdżenia dynamicznego zagnieżdżenie nie wynika wprost ze struktury kodu źródłowego. Sterowanie może zostać przekazane z wnętrza bloku `try..endtry` do zupełnie innego miejsca programu, niżby to wynikało z samej struktury kodu. Jeśli w tamtym miejscu realizowany jest blok chroniony instrukcji `try..endtry`, to mamy do czynienia z zagnieżdżeniem dynamicznym. Zagnieżdżenie dynamiczne można zrealizować na kilka sposobów, a najbardziej chyba intuicyjnym (i zapewne znanym Czytelnikowi z języków wysokiego poziomu) z nich jest wywołanie

[12] Chyba że sterowanie zostanie wcześniej przekazane z powrotem do bloku `try..endtry`, na przykład kiedy jest on umieszczony w ciele pętli.

[13] Zagnieżdżenie statyczne należy rozumieć jako „fizyczne”, bo wynikające wprost ze struktury kodu źródłowego, umieszczenie jednego bloku wewnątrz drugiego. Kiedy mowa o zagnieżdżaniu instrukcji, zwykle chodzi o zagnieżdżanie statyczne.

procedury. Kiedy więc w dalszej części książki omawiany będzie sposób pisania i wywoływania procedur w języku asemblerowym, należy pamiętać, że wywołanie procedury z bloku kodu chronionego może doprowadzić do dynamicznego zagnieżdżenia bloków `try..endtry` — wystarczy, aby procedura również zawierała blok `try..endtry` (i aby został on wykonany).

1.11.2. Klauzula unprotected bloku try..endtry

Wykonywanie instrukcji `try` powoduje zawsze zachowanie bieżącego środowiska wyjątków i przygotowanie systemu wykonawczego (na wypadek zgłoszenia wyjątku) do przekazania sterowania do odpowiedniej klauzuli `exception`. Jeśli program przebrnie przez blok instrukcji chronionych, pierwotne środowisko wyjątków jest przywracane, a sterowanie przekazywane jest do pierwszej instrukcji znajdującej się za klauzulą `endtry`. Bardzo ważną rolę w tej procedurze ma ostatni jej etap, czyli przywrócenie pierwotnego środowiska wyjątków. W przypadku jego pominięcia wszelkie kolejne wyjątki powodowałyby przekazanie sterowania do opuszczonego już bloku `try..endtry`. Problem ten ilustruje listing.1.10.

Listing 1.10. *Niepoprawne opuszczenie bloku instrukcji try..endtry*

```
program testBadInput4;
#include( "stdlib.hhf" )

static
    input:      uns32;

begin testBadInput4;

    // Poniższa pętla jest kontynuowana dopóty, dopóki użytkownik wprowadza
    // nieprawidłową liczbę; wyjście z pętli następuje w wyniku wykonania instrukcji break.

    forever

        try

            stdout.put( "Podaj liczbe calkowita: " );
            stdin.get( input );
            stdout.put( "Wprowadzono: ", input, nl );
            break;

        exception( ex.ValueOutOfRange )

            stdout.put( "Liczba jest zbyt duza; powtorz wprowadzanie." nl );

        exception( ex.ConversionError )

            stdout.put( "Ciag zawiera niedozwolone znaki; powtorz wprowadzanie." nl
);

        endtry;

    endfor;
```

```
    // Uwaga: poniższy kod znajduje się poza pętlą i nie jest osadzony w bloku chronionym
    // instrukcji try..endtry

    stdout.put( "Podaj jeszcze jedna liczbe: " );
    stdin.get( input );
    stdout.put( "Nowa liczba to:", input, nl );

end testBadInput4;
```

Niniejszy przykład z pozoru implementuje niezawodny, odporny na błędy system wprowadzania danych: wokół bloku kodu chronionego rozciągnięta jest pętla wymuszająca ponowne wprowadzenie danych przez użytkownika, jeśli w poprzedniej próbie nie udało się ich poprawnie odczytać. Pomysł jest dobry; problem tkwi w jego nieprawidłowej realizacji. Otóż instrukcja `break` powoduje opuszczenie pętli `forever..endfor` bez przywrócenia poprzedniego stanu środowiska wyjątków. Z tego względu, kiedy program wykonuje procedurę `stdin.get` umieszczoną poza pętlą (a tym bardziej poza blokiem kodu chronionego), system obsługi wyjątków wciąż traktuje jej kod jako chroniony. Jeśli więc przy okazji jego wykonania zgłoszony zostanie wyjątek, sterowanie zostanie przekazane z powrotem do stosownej klauzuli `exception` opuszczonego już przecież bloku `try..endtry`. Jeśli identyfikatorem wyjątku będzie `ex.ValueOutOfRange` albo `ex.ConversionError`, wyświetlony zostanie stosowny komunikat, a użytkownik **ponownie zmuszony będzie do wprowadzenia liczby**. Tego zaś programista raczej nie przewidywał.

Przekazanie sterowania do nieodpowiedniego bloku `try..endtry` to tylko część problemu. Kod z listingu 1.10 obarczony jest inną poważną wadą, związaną ze sposobem zachowywania i następnie przywracania przez system wykonawczy HLA środowiska wyjątków. W szczególności bowiem HLA zachowuje informacje o stanie systemu wyjątków w specjalnym obszarze pamięci zwanym **stosem**. Jeśli po wyjściu z bloku `try..endtry` stan środowiska wyjątków nie zostanie przywrócony, informacje o tym środowisku pozostaną na stosie; obecność na nim nadmiarowych danych może zaś doprowadzić do błędnego działania programu.

Jest już chyba jasne, że program nie powinien opuszczać bloku `try..endtry` z pominięciem etapu przywrócenia pierwotnego stanu środowiska wyjątków. Wiadomo, że sposób wprowadzania danych zaprezentowany na listingu 1.10 jest niepoprawny, ale z drugiej strony sama koncepcja wymuszania powtarzania operacji wprowadzania aż do skutku jest jak najbardziej słuszna. Można ją zaimplementować przy użyciu specjalnej klauzuli bloku `try..endtry`. Spójrzmy na kod z listingu 1.11.

Listing 1.11. *Blok pozbawiony ochrony wewnątrz bloku try..endtry*

```
program testBadInput5;
#include( "stdlib.hhf" )

static
    input:    uns32;

begin testBadInput5;
```

```
    // Poniższa nieskończona pętla wykonywana jest dopóty, dopóki użytkownik wprowadza
    // niepoprawne dane. Wyjście z pętli następuje w wyniku wykonania instrukcji break.
    // Instrukcja ta, mimo że obecna wewnątrz bloku try..endtry, znajduje się w bloku
    // kodu niechronionego sygnalizowanego klauzulą unprotected.

    forever

        try

            stdout.put( "Podaj liczbe calkowita: " );
            stdin.get( input );
            stdout.put( "Wprowadzono: ", input, nl );

        unprotected

            break;

        exception( ex.ValueOutOfRange )

            stdout.put( "Liczba jest zbyt duza; powtorz wprowadzanie." nl );

        exception( ex.ConversionError )

            stdout.put( "Ciag zawiera niedozwolone znaki; powtorz wprowadzanie." nl
);

        endtry;

    endfor;

    // Uwaga: poniższy kod znajduje się poza pętlą i nie jest osadzony w bloku chronionym
    // instrukcji try..endtry

    stdout.put( "Podaj jeszcze jedna liczbe: " );
    stdin.get( input );
    stdout.put( "Nowa liczba to:", input, nl );

end testBadInput5;
```

Osiągnięcie klauzuli `unprotected` inicjuje operację przywrócenia pierwotnego stanu środowiska wyjątków. Oczywiste jest, że po przejściu do bloku kodu niechronionego wykonanie kodu nie jest już chronione na wypadek zgłoszenia wyjątków. Klauzula `unprotected` nie znosi jednak ochrony kodu realizowanej w dynamicznie zagnieżdżonych zewnętrznych blokach `try..endtry` — klauzula `unprotected` odnosi się jedynie do tego bloku `try..endtry`, w którym została umieszczona. Jako że na listingu 1.11 instrukcja przerywająca pętlę, `break`, znajduje się za klauzulą `unprotected`, przekazanie sterowania poza pętlę odbywa się po uprzednim przywróceniu stanu wyjątków.

Słowo `unprotected`, jeśli już występuje w ramach instrukcji `try`, powinno znajdować się bezpośrednio za blokiem kodu chronionego. Musi więc poprzedzać wszystkie klauzule `exception`.

W obliczu zgłoszenia wyjątku system wykonawczy HLA automatycznie przywraca pierwotny stan środowiska wykonania. Z tego względu w ramach kodu obsługi wyjątku, po klauzuli `exception` można swobodnie korzystać z instrukcji `break`, nie obawiając się o negatywne efekty charakterystyczne dla niepoprawnego opuszczenia bloku `try..endtry`.

Z racji przywrócenia pierwotnego stanu wyjątków w bloku niechronionym oraz w kodzie obsługi wyjątku po klauzuli `exception`, wystąpienie wyjątku w jednym z tych obszarów powoduje natychmiastowe przekazanie sterowania do zewnętrznego bloku `try..endtry`, w którym blok bieżący został dynamicznie zagnieżdżony. W przypadku braku takiego bloku program jest awaryjnie przerywany, a na standardowe wyjście wyprowadzany jest stosowny komunikat o błędzie.

1.11.3. Klauzula anyexception bloku try..endtry

Typowo instrukcji `try` towarzyszy szereg klauzul `exception` obsługujących wszelkie wyjątki, których wystąpienie w kodzie chronionym zdoła przewidzieć programista. Niekiedy to nie wystarcza — konieczne jest wtedy zapewnienie obsługi wszystkich możliwych (a nie tylko przewidzianych) wyjątków, aby program nie został przedwcześnie zakończony. Jeśli programista jest autorem całości kodu bloku chronionego, powinien być zdolny do przewidzenia wszystkich sytuacji wyjątkowych związanych potencjalnie z wykonaniem bloku. Jeśli jednak w bloku tym znajdują się wywołanie procedury bibliotecznej, wywołanie funkcji interfejsu systemu operacyjnego czy inne instrukcje, nie w pełni kontrolowane przez programistę, przewidzenie wszystkich możliwych wyjątków może być niemożliwe. Tymczasem zgłoszenie wyjątku o identyfikatorze innym niż identyfikatory podane w kolejnych klauzulach `exception` może doprowadzić do załamania programu. Szczęśliwie język HLA przewiduje umieszczenie w bloku `try..endtry` klauzuli `anyexception`, do której dopasowywane są wszystkie wyjątki, których identyfikatorów nie uda się odpasować do klauzul `exception`.

Klauzula `anyexception` nie różni się wiele od klauzuli `exception` — jedyna odmienność tkwi w braku konieczności określania identyfikatora wyjątku (to oczywiste). Jeśli obok klauzul `exception` w bloku `try..endtry` znajduje się klauzula `anyexception`, powinna ona być ostatnią klauzulą obsługi wyjątków w bloku. Dopuszczalne jest naturalnie, aby klauzula `anyexception` była jedyną klauzulą obsługi wyjątków w bloku `try..endtry`.

Po przekazaniu sterowania do klauzuli `anyexception` identyfikator wyjątku umieszczany jest w rejestrze EAX. Identyfikator ten można w ramach kodu obsługi wyjątku sprawdzać, dopasowując obsługę do przyczyny zgłoszenia wyjątku.

1.11.4. Instrukcja try..endtry i rejestry

Każdorazowe wkroczenie do bloku `try..endtry` oznacza konieczność zachowania na stosie 16 bajtów informacji o stanie środowiska wyjątków. Przywrócenie środowiska wyjątków po opuszczeniu tego bloku (albo w wyniku osiągnięcia klauzuli `unprotected`) wymaga odczytania właśnie owych 16 bajtów. Do momentu zgłoszenia wyjątku realizowanie kodu chronionego nie wpływa na zawartość żadnych rejestrów. Sytuacja zmienia się w momencie zgłoszenia wyjątku w wyniku wykonania instrukcji kodu chronionego.

Po przejściu do klauzuli `exception` rejestr EAX zawiera identyfikator (numer) wyjątku. Reszta rejestrów ogólnego przeznaczenia zawiera wartości nieokreślone. Ponieważ system operacyjny może zgłosić wyjątek w reakcji na błąd sprzętowy, nie powinno się nawet zakładać, że rejestry ogólnego przeznaczenia zawierać będą po rozpoczęciu obsługi wyjątku wartości, które znajdowały się w nich pierwotnie. Kod generowany przez system HLA do obsługi wyjątków może być różny w różnych wersjach kompilatora, więc poleganie na zawartości rejestrów w kodzie obsługi wyjątków jest co najmniej ryzykowne.

Jako że po przejściu do bloku kodu obsługi wyjątku nie można czynić żadnych założeń co do zawartości rejestrów (z wyjątkiem rejestru EAX), to jeśli kod kontynuowany za klauzulą `endtry` zakłada obecność w rejestrach jakichś konkretnych wartości (np. wartości umieszczonych tam przed wkroczeniem do bloku kodu chronionego), należy te wartości przywrócić samodzielnie. Zaniedbanie tego może doprowadzić do błędnego działania programu, przy czym błędy tego rodzaju są tym trudniejsze do wykrycia, że sytuacje wyjątkowe z definicji zdarzają się rzadko, co utrudnia odtworzenie i diagnostykę błędu; dodatkowym utrudnieniem jest to, że nie zawsze zgłoszenie wyjątku musi zmienić wartość konkretnego rejestru. Sposób rozwiązania problemu zachowania wartości rejestrów prezentuje następujący fragment kodu:

```
static
    sum:    int32;
…

    mov( 0, sum );
    for( mov( 0, ebx ); ebx < 8; inc( ebx ) ) do

        push( ebx );      // Zachowanie wartości rejestru EBX na wypadek wyjątku.
        forever
            try

                stdin.geti32();
                unprotected break;

            exception( ex.ConversionError )

                stdout.put( "Niedozwolona wartość; sprobuj ponownie: " );

            endtry;
        endfor;
        pop( ebx );     // Przywrócenie zawartości rejestru EBX.
        add( ebx, eax );
        add( eax, sum );

endfor;
```

Ponieważ mechanizm obsługi wyjątków powoduje potencjalne zmiany wartości rejestrów i ponieważ obsługa wyjątków to proces stosunkowo nieefektywny, nie powinno się stosować instrukcji `try` w roli zwykłej struktury sterującej wykonaniem programu (na przykład symulując w bloku `try..endtry` działanie instrukcji wyboru znanych z języków wysokiego poziomu, jak `case` czy `switch`). Takie praktyki wpływają ujemnie na wydajność programu; mogą tez wprowadzać niepożądane, trudne do wykrycia i zdiagnozowania efekty uboczne wynikające z zakłócania zawartości rejestrów.

Działanie mechanizmu wyjątków opiera się na założeniu, że rejestr EBP wykorzystywany jest wyłącznie w roli wskaźnika **rekordów aktywacji** (rekordy aktywacji omawiane są w rozdziale poświęconym procedurom). Standardowo programy HLA wykorzystują ów rejestr właśnie w tej roli. Warunkiem poprawnego działania programów jest więc unikanie modyfikowania zawartości rejestru EBP. Jeśli rejestr ten zostanie użyty w roli rejestru ogólnego przeznaczenia, na przykład w obliczeniach arytmetycznych, obsługa wyjątków systemu wykonawczego HLA nie będzie działać poprawnie, pojawić się też mogą dodatkowe problemy. To samo tyczy się zresztą rejestru ESP — jego również nie należy wykorzystywać jako rejestru ogólnego przeznaczenia.