

DLA ATARI®

**Action!**<sup>TM</sup>

**Potężny!**

**Elastyczny!**

**Szybki!**



Precision  
Software Tools

FOR ATARI®

**Action!**<sup>TM</sup>

**Powerful!**

**Flexible!**

**Fast!**



<sup>TM</sup>

**Precision  
Software Tools**

# SYSTEM ACTION!

Konwersji z OCR Ksero podjął się Tatko74 w 2022r.

OCR do wersji edytowalnej .doc - LOKR - 04.2024r.

Źródło [atarionline.pl](http://atarionline.pl)

## **ACTION!**

Strukturalny język programowania będący kombinacją cech języków: C, Pascal, Ada, Basic. Do języka dołączony jest edytor, kompilator, monitor oraz biblioteka podprogramów.



*Swoją pracę dedykuję niezwykłemu przyjacielowi Marcinowi,  
dzięki któremu wiedziałem, że Atari to nie konkurencja dla mojego C64,  
ale zupełnie inne możliwości dla młodego człowieka.  
K. L.*

# **ACTION!**

<b>CZĘŚĆ I</b>	<b>WSTĘP</b>	<b>1</b>
1	System <i>ACTION!</i>	2
2	Jak napisać i uruchomić program	3
<b>CZĘŚĆ II</b>	<b>EDYTOR ACTION!</b>	<b>4</b>
1	Wstęp	5
2	Komendy edytora	7
3	Porównanie edytorów <i>ACTION!</i> i Atari	14
4	Uwagi techniczne	16
<b>CZĘŚĆ III</b>	<b>MONITOR ACTION!</b>	<b>17</b>
1	Wstęp	18
2	Komendy monitora	19
3	Możliwości tekstowania programu	24
<b>CZĘŚĆ IV</b>	<b>JĘZYK PROGRAMOWANIA ACTION!</b>	<b>25</b>
1	Wstęp	27
2	Słowa kluczowe	28
3	Podstawowe typy danych	29
4	Wyrażenia	32
5	Instrukcje	37
6	Procedury i funkcje	48
7	Dyrektywy kompilatora	58
8	Złożone typy danych	60
9	Dodatkowe możliwości programowania	72
<b>CZĘŚĆ V</b>	<b>KOMPILATOR ACTION!</b>	<b>76</b>
1	Wstęp	77
2	Sposób działania kompilatora - przydzielenie pamięci	78
3	Korzystanie z menu opcji	80
4	Uwagi techniczne	81
<b>CZĘŚĆ VI</b>	<b>BIBLIOTEKA PODPROGRAMÓW ACTION!</b>	<b>82</b>
1	Wstęp	84
2	Podprogramy wyrowadzające dane	85
3	Podprogramy wprowadzania danych	88
4	Podprogramy manipulujące plikami	90
5	Grafika i manipulatory gier	92
6	Operacje na łańcuchach tekstowych	97
7	Podprogramy różne	99
<b>DODATKI</b>		<b>103</b>
A	Składnia języka <i>ACTION!</i>	103
B	Mapa pamięci	107
C	Komunikaty błędów	108
D	Bibliografia	109
E	Zbiór komend edytora <i>ACTION!</i>	110
F	Zbiór komend monitora <i>ACTION!</i>	112
G	Menu opcji	113
H	„PRIMES” Benchmark	114
I	Przenoszenie instrukcji BASIC do programu <i>ACTION!</i>	115

# Część I System *ACTION!*

Rozdział 1	Wstęp	1
Rozdział 2	System <i>ACTION!</i>	2
Rozdział 3	Jak napisać i uruchomić program	3

## Rozdział 1 Wstęp

W porównaniu z językiem Basic, *ACTION!* jest dużo szybszy, ma lepszy edytor, a jest równie prosty do nauczenia. W stosunku do asemblera jest on prawie tak samo szybki, natomiast pisanie programu trwa znacznie krócej ze względu na większą naturalność języka, jego edytor i bibliotekę.

Dla tych, dla których *ACTION!* będzie pierwszym językiem programowania, który poznają, proponujemy aby czytali ten podręcznik bardzo dokładnie i nie przechodzili do dalszych części materiału bez zrozumienia wcześniejszych.

Zwracamy na to uwagę, ponieważ podręcznik ten nie był pisany z myślą o nauczaniu języka lecz ma na celu raczej poinformowanie użytkownika o możliwościach całego systemu.

Nie oznacza to wcale, że nie zrozumiecie tego co przeczytacie (wręcz przeciwnie). Pisząc to mieliśmy tylko na myśli fakt, że podręcznik nie omawia wszystkich możliwości języka.

Oczekujemy na waszą pomysłowość i dociekliwość oraz wierzymy, że odkryjecie wiele możliwości, o których my nawet nie marzyliśmy...

## UWAGI O PODRĘCZNIKU

Podręcznik jest podzielony na sześć części oraz zawiera kilka dodatków. Każda z części osobno omawia jedną stronę systemu *ACTION!*. Każda część jest poprzedzona spisem treści, wstępem oraz spisem używanych terminów. Słabą stroną tego podziału jest to, że trzeba czytać wszystko o jednej części systemu nie wiedząc nic o innych. Aby uniknąć tego, proponujemy przeczytać wcześniej wstęp do każdej części. Ostatni rozdział części I podręcznika zawiera informacje jak współpracują poszczególne części systemu *ACTION!*.

## Rozdział 2                    **SYSTEM ACTION!**

System *ACTION!* składa się z pięciu różnych części:

- monitor
- edytor
- język programowania
- kompilator
- biblioteka

Monitor zarządza całym systemem. Poprzez niego można wywołać edytor kompilator lub uzyskać dostęp do niektórych opcji systemu.

Edytor używany jest do tworzenia nowych programów lub modyfikacji starych. Jest to zwykły edytor tekstowy i nie zawiera informacji o składni języka *ACTION!* (można go więc używać do innych zastosowań). Edytor ten pozwala także odczytać tekst z urządzeń peryferyjnych i umieścić go w pamięci komputera.

Program napisany w języku *ACTION!* jest translowany poprzez kompilator na formę zrozumiałą dla komputera (język maszynowy). Dopiero wówczas program może być uruchomiony.

„Po co tak skomplikowany proces? Basic działa na innej zasadzie.”

Po pierwsze, po zrozumieniu istoty tego procesu okaże się on wcale nie tak bardzo skomplikowany. Po drugie, Basic rzeczywiście działa na innej zasadzie ponieważ posiada interpreter, a nie kompilator.

BASIC tłumaczy każdą linię programu podczas jego wykonywania. Zajmuje to oczywiście trochę czasu, a więc czas wykonania programu wydłuża się.

*ACTION!* rozdziela od siebie fazę sprawdzania składni i fazę wykonania programu. Kompilator sprawdza program pod względem poprawności składni, a następnie tłumaczy go na język maszynowy.

Po wykonaniu tych czynności program może być natychmiast uruchomiony (składnia nie będzie już sprawdzana). Spowoduje to znaczne przyspieszenie jego wykonania.

Jak już zauważono, kompilator *ACTION!* tłumaczy program napisany w języku *ACTION!* na kod maszynowy. Jedyną rzeczą jaka jest wymagana jest to aby program był napisany w poprawnej formie. Jeżeli składnia, która została użyta jest nielegalna, kompilator zakomunikuje o błędzie.

System *ACTION!* zawiera także grupę podprogramów standardowych, które można użyć w swoim programie. Zbiór tych podprogramów nazwany został biblioteką *ACTION!* i umożliwia nie tylko wykonywanie takich instrukcji jak w Basicu instrukcje PLOT, DRAWTO, PRINT itp. lecz również wiele innych, bez konieczności pisania swoich podprogramów.

### **UWAGI TECHNICZNE**

Kompilator *ACTION!* tłumaczy program na język maszynowy 6502.

## Rozdział 3 Jak napisać i uruchomić program

Rozdział ten ma na celu bliższe zapoznanie Was z systemem *ACTION!* Aby tego dokonać, napiszemy krótki program pod edytorem, skompilujemy go, a następnie uruchomimy.

Po załadowaniu systemu *ACTION!* następuje zgłoszenie edytora. Dzięki temu program może być od razu wprowadzany do komputera. W rozdziale tym przedstawimy tylko kilka cech i komend edytora. Pozostałe będą omówione dokładnie w części II tego podręcznika.

Wprowadź do komputera tekst programu zamieszczony poniżej:

```
PROC hello()
    PrintE („Hello World”)
RETURN
```

Zanim skompilujemy ten program, wcześniej omówimy co on ma „robić”. Instrukcje `PROC` i `RETURN` są wymagane przez język *ACTION!* do oznaczenia początku i końca procedury. Program napisany w języku *ACTION!* zbudowany jest właśnie z szeregu takich podprogramów nazwanych procedurami i funkcjami. Pozwala to pisać programy częściami, koncentrując się w danej chwili tylko na jednej z nich. Programy napisane w ten sposób są również łatwiejsze do odszyfrowania przez innych.

Powyższą procedurą została nazwana „hello” ponieważ jej wykonanie spowoduje wyświetlenie na ekranie napisu „Hello World”.

Instrukcja `PrintE` jest biblioteczną procedurą standardową i ma na celu wyprowadzić na ekran napis ujęty w nawiasy. Wywołanie tej procedury jest jedyną instrukcją w procedurze „hello”.

Po wprowadzeniu z klawiatury tekstu programu zostaje on umieszczony w buforze edytora. Aby skompilować program, a następnie go uruchomić konieczne jest wyjście z edytora i przejście na poziom monitora. Można tego dokonać poprzez wykonanie komendy edytora `<CTRL> <SHIFT> M`.

Dla sprawdzenia składni programu i przetłumaczenia go na język maszynowy należy z poziomu monitora wywołać kompilator. Nastąpi to po wprowadzeniu komendy „`COMPILE <RETURN>`”.

Jeżeli kompilator wykryje błędy składni, zostaje wydrukowany komunikat o błędzie i sterowanie powraca do poziomu monitora. Również w przypadku gdy żaden błąd nie zostanie wykryty, sterowanie powraca na poziom monitora.

Z poziomu tego można uruchomić już skompilowany program przez wprowadzenie komendy „`RUN <RETURN>`”. Po wykonaniu programu ekran powinien mieć następujący wygląd:



W ten sposób napisaliście swój pierwszy program w języku *ACTION!*.

Jeżeli kompilator zasygnalizował Wam, że w programie jest błąd, oznacza to, że nie dość dokładnie go wprowadzaliście. Należy dostać się z powrotem do edytora i poprawić błąd. Zwróćcie uwagę, że kursor jest umieszczony na pozycji w której kompilator odnalazł błąd. Nie jest więc konieczne przeglądanie całego programu.

Po poprawieniu błędu program należy ponownie skompilować, a następnie uruchomić go.

**UWAGA:** Lista kodów błędów wraz z ich opisem znajduje się w dodatku C.

# Część II      Edytor *ACTION!*

<b>Rozdział 1    Wstęp</b>		<b>5</b>
1.1	Notacja	5
1.2	Cechy i możliwości edytora	5
<b>Rozdział 2    Komendy edytora</b>		
2.1	Wejście na poziom edytora	7
2.2	Wyjście z edytora	7
2.3	Wprowadzanie tekstu	7
2.3.1	Pliki tekstowe I/O	7
2.3.2	Ustawianie długości linii	8
2.4	Sterowanie kursorem	8
2.4.1	Tabulatory	8
2.4.2	Przeszukiwanie tekstu	9
2.5	Poprawianie tekstu	9
2.5.1	Kasowanie znaków	9
2.5.2	Wstawianie/zamiana znaków	9
2.5.5	Kasowanie całej linii	10
2.5.4	Wstawianie linii	10
2.5.5	Dzielenie i łączenie linii	10
2.5.6	Podstawianie tekstu	10
2.5.7	Odzyskiwanie skasowanego tekstu	11
2.6	Okna tekstowe	11
2.6.1	Sterowanie oknem tekstowym	11
2.6.2	Tworzenie drugiego okna	11
2.6.3	Zamiana okna aktualnego	12
2.6.4	Kasowanie zawartości okna	12
2.6.5	Kasowanie okna	12
2.7	Przesuwanie/kopiowanie bloków tekstu	12
2.8	Etykiety	13
<b>Rozdział 3    Porównanie edytorów ACTION! i ATARI</b>		<b>14</b>
3.1	Komendy wspólne	14
3.2	Komendy różne dla obydwóch edytorów	14
3.3	Komendy unikalne dla edytora ACTION!	15
<b>Rozdział 4    Uwagi techniczne</b>		<b>16</b>
4.1	Pliki tworzone przez inne edytory tekstu	16
4.2	Błąd „przepełnienia pamięci”	16



# Rozdział 1      Wstęp

Edytora używa się podczas tworzenia nowego programu lub przy poprawianiu starego. Jeśli używaliście już jakiegoś edytora zauważycie, że edytor *ACTION!* jest bardziej rozbudowany. Ze względu na jego możliwości można by go nazwać właściwie procesorem tekstu.

Mimo, że ma on tak duże możliwości, praca z nim jest stosunkowo prosta. Jak już wspomniano, edytor *ACTION!* może być wykorzystywany do innych celów, nie tylko do pisania programów. Można go używać do tworzenia dowolnego tekstu np. listu.

## 1.1      Notacja

Komendy i znaki specjalne muszą być ujęte w apostrofy `'/'`.

Użycie konkretnego klawisza z klawiatury Atari będzie sygnalizowane przez znaki „<” i „>”, np.: `<BACK S>`. Niektóre z klawiszy mają więcej niż jedną etykietę. W takich przypadkach będzie użyta ta, która lepiej opisuje daną komendę edytora. Litery (A-Z) oraz cyfry (0-9) nie będą zamykane w nawiasy ostre.

Niektóre z komend edytora wymagają aby nacisnąć więcej niż jeden klawisz jednocześnie. W takich przypadkach klawisze które należy przycisnąć będą podane jeden za drugim, w odpowiedniej kolejności. Np. napis:

```
<SHIFT> <DELETE>
```

będzie oznaczał, że powinno się wcisnąć klawisz: oznaczony przez `<SHIFT>`, a następnie przez `<DELETE>`. Podczas używania edytora dolna linia ekranu jest zarezerwowana na komunikaty. Zwykle znajduje się tam napis:

```
ACTION! (c) 1983 ACS
```

Niektóre z komend edytora używające linii na komunikaty do wymiany informacji z użytkownikiem pamiętają informację jaka im się podało poprzednio. W takich przypadkach jeżeli chce się podać ponownie tą samą odpowiedź, wystarczy nacisnąć klawisz `<RETURN>`. Pozwala to zaoszczędzić czas ponieważ nie trzeba wprowadzać tej samej odpowiedzi wiele razy. Jeżeli chce się zmienić odpowiedź można wprowadzić nową lub zmienić tylko część poprzedniej.

## 1.2      Cechy i możliwości edytora

### OKNA TEKSTU

Patrząc na telewizor lub monitor wyobraźcie sobie, że patrzycie przez okno. W danej chwili możecie zobaczyć tylko 23 linie po 38 znaków w każdej. Byłoby to dużym ograniczeniem gdyby nie możliwość przesuwania okna. Edytor *ACTION!* umożliwi przesuwanie, okna tak w pionie, jak i w poziomie i w rezultacie można obejrzeć cały program.

Jeżeli w całym tekście tylko jedna linia przekracza granice ekranu, nie jest konieczne przesuwanie całego okna lecz można przesunąć tylko tą jedną żadaną linię.

Edytor daje możliwość rozbić ekran na dwa oddzielne okna, którymi można sterować oddzielnie. Pozwala to przeglądać dwa różne programy lub różne części jednego programu w tym samym czasie.

### LINIE TEKSTU

Edytor *ACTION!* został tak opracowany aby czytanie programu było jak najprostsze. Dopuszczalna długość linii wynosi 240 znaków (mimo, że okno ukazuje tylko 38 znaków jednocześnie). Mając taką możliwość można w programach, dla poprawienia ich czytelności, robić wcięcia, nie martwiąc się, że zostanie przekroczona maksymalna długość linii. Edytor dopuszcza również na umieszczanie pustych linii pomiędzy częściami programu, które użytkownik chce od siebie oddzielić.

**UWAGA:** Użytkownik ma możliwość ustalenia maksymalnej długości linii (np. może ją dobrać do ilości znaków na drukarce). Edytor sygnalizuje dojście do końca linii dźwiękiem.

**UWAGA:** Jeżeli linia tekstu jest dłuższa niż okno, znaki na granicy o okna są wyświetlane na odwróconym tle. Ma to informować użytkownika, że dana linia zawiera więcej niż 38 znaków.

### **WYSZUKIWANIE I PODSTAWIANIE**

Edytor umożliwia poszukiwanie konkretnego łańcucha tekstu i ustawia kursor na pierwszy znaleziony w programie odpowiadający mu zestaw znaków. Dodatkowo, w tym samym rozkazie, można nakazać aby odnaleziony łańcuch został zastąpiony wyspecyfikowanym przez użytkownika tekstem.

### **PRZESUWANIE BLOKÓW TEKSTU**

Czy mieliście już kiedyś sytuację, że po wprowadzeniu programu okazało się, że pewna grupa linii powinna być przesunięta w inne miejsce? W *ACTION!* jest to proces błyskawiczny. W tym celu należy przejściowo umieścić określone linie (blok tekstu) w buforze kopii. Następnie kursor przesuwa się w miejsce gdzie linie te mają się znaleźć i "dokleja" tam zawartość bufora. Istnieje możliwość kopiowania tego samego zestawu linii w kilku miejscach.

### **STEROWANIE KURSOREM**

Kursor jest sterowany z klawiatury takimi klawiszami jak <CTRL>, <UP arrow> itp. Istnieje również możliwość jego przesuwania w określone miejsce tekstu poprzez użycie komendy „FIND” lub mechanizmu etykiet.

### **ETYKIETY**

W tekście można umieścić niewidoczny oznacznik miejsca, który będziemy nazywali etykietą. Edytor umożliwia następnie prostą komendą przesunięcie kursora w to oznaczone miejsce.

Liczba etykiet jest ograniczona jedynie ilością klawiszy na klawiaturze komputera. Wynika to z tego, że każda etykieta jest pojedynczym znakiem.

## Rozdział 2 Komendy edytora

Rozdział ten został poświęcony komendom edytora. Wykonanie każdej komendy można przerwać przez wciśnięcie klawisza <ESC>. Pełna lista komend edytora *ACTION!* została zamieszczona w dodatku E.

### 2.1 Wejście na poziom edytora

Po wprowadzeniu do komputera całego systemu użytkownik automatycznie znajduje się w obszarze: działania edytora i zbędne są dodatkowe instrukcje.

Jeżeli opuści się system *ACTION!* i wejdzie pod DOS (OS/A+, DOS XL lub Atari DOS), a następnie ponownie wprowadzi się *ACTION!*, użytkownik znajdzie się na poziomie monitora systemu. Aby znaleźć się pod edytorem należy wówczas wprowadzić komendę:

```
E <RETURN>
```

### 2.2 Wyjście z edytora

Jedynym sposobem wyjścia z edytora (z wyjątkiem wyłączenia komputera) jest wprowadzenie komendy:

```
<CTRL> <SHIFT> M
```

Komenda ta spowoduje wejście na poziom monitora, skąd można wywołać inne części systemu *ACTION!* lub go opuścić i wejść pod DOS.

### 2.3 Wprowadzanie tekstu

Tekst wprowadza się w taki sposób jak na zwykłej maszynie do pisania. Nie są wymagane żadne dodatkowe komendy. Każdorazowo po osiągnięciu końca linii edytor reaguje sygnałem dźwiękowym. Więcej informacji na ten temat znajduje się w §2.3.2.

Jeżeli chce się wprowadzić znaki sterujące jako tekst należy wcześniej nacisnąć <ESC>. Pozwoli to edytorowi rozpoznać, w których przypadkach znaki sterujące mają być traktowane jako tekst, a nie jako komendy.

Poprawianie wcześniej wpisanego tekstu może odbywać się w dwóch trybach: „ZAMIANA” i „WSTAWIANIE”.

W trybie „ZAMIANA” nowo wprowadzony tekst zastępuje znak po znaku tekst poprzedni.

W trybie „WSTAWIANIE” nowo wprowadzony tekst będzie umieszczony w miejscu, gdzie znajduje się kursor, natomiast cały poprzedni tekst zostanie przesunięty o tyle pozycji ile zostało wprowadzonych nowych znaków.

Komenda <CTRL> <SHIFT> I pozwala na zmianę jednego trybu na drugi.

Po użyciu tej komendy aktualny tryb pracy zostanie wyświetlony w linii na komunikaty (więcej informacji na ten temat znajduje się w §2.5.2.).

**UWAGA:** Przy pierwszym wejściu pod edytor automatycznie ustawiony jest tryb „ZAMIANA”.

Jeżeli chce się skasować cały tekst w pliku, należy umieścić kursor w oknie które go zawiera i nacisnąć <SHIFT> <CLEAR>. Spowoduje to wymazanie nie tylko tekstu z obrębu okna, ale całego pliku do którego ten tekst należy.

#### 2.3.1 Pliki tekstowe I/O

Edytor umożliwia zapamiętanie, a następnie odczyt plików tekstowych, w pamięci zewnętrznej (dyskietka, kaseta magnetofonowa itp.).

Aby umieścić program w buforze edytora należy ustawić kursor w oknie, w którym widoczny jest dany plik. Jeżeli używa się tylko jednego okna, nie jest to konieczne. Następnie należy wprowadzić komendę:

```
<CTRL> <SHIFT> W
```

W linii komunikatów pojawi się napis:

```
Write?
```

W odpowiedzi należy wpisać nazwę jaką ma mieć program i nacisnąć <RETURN>. Nazwa pliku musi być zgodna z DOS-em, którego się używa. Jeżeli DOS nie jest używany, nazwa pliku będzie się składała tylko ze znaku reprezentującego urządzenie (C - kaseta magnetofonowa, P - drukarka itp.) oraz dwukropka.

Aby odczytać plik do bufora edytora, należy przesunąć kursor do linii poprzedzającej miejsce, gdzie ma być on umieszczony, a następnie wprowadzić komendę:

```
<CTRL> <SHIFT> R
```

W linii komunikatów pojawi się napis:

```
Read?
```

W odpowiedzi na to pytanie należy wprowadzić nazwę pliku, który się chce odczytać (na tej samej zasadzie jak przy zapisie).

Jeżeli używa się dyskietek, istnieje możliwość odczytania katalogu danej dyskietki poprzez wprowadzenie:

```
Read? ?1:*.*
```

Zostanie odczytany katalog dyskietki znajdującej się w mechanizmie dyskowym nr 1. Jeśli chce się odczytać katalog dyskietki znajdującej się w innym mechanizmie, należy zamiast "1" wprowadzić jego numer. Możliwość ta jest bardzo użyteczna ponieważ nie jest konieczne przejście pod DOS aby dowiedzieć się co znajduje się na danej dyskietce.

## 2.3.2 Ustawianie długości linii

Jak wspomniano już w części I rozdział 2.3. istnieje możliwość ustawienia maksymalnej długości linii. Czynność ta zostanie dokładnie omówiona w części III podręcznika w rozdziale 2.5.

## 2.4 Sterowanie kursorem

Aby przesunąć kursor o jeden znak w lewo należy wprowadzić:

```
<CTRL> <left arrow>
```

Aby przesunąć kursor o jedną pozycję w prawo należy nacisnąć:

```
<CTRL> <right arrow>
```

W celu przesunięcia kursora o jedną linię w górę, należy nacisnąć:

```
<CTRL> <up arrow>
```

natomiast o jedną linię w dół:

```
<CTRL> <down arrow>
```

Komendy opisane powyżej są zwykłymi komendami edytora ekranu komputera Atari wykonywanymi wprost z klawiatury.

Dla przyspieszenia procesu pisania programu, edytor *ACTION!* dopuszcza inne sposoby zmiany położenia kursora.

Aby przesunąć kursor bezpośrednio na początek linii, w której się znajduje, należy wprowadzić:

```
<CTRL> <SHIFT> <
```

natomiast aby znalazł się na końcu linii:

```
<CTRL> <SHIFT> >
```

W wyniku działania tych komend kursor znajdzie się odpowiednio na początku lub na końcu linii, nawet w przypadku gdy nie będzie on widoczny w oknie ekranu. Linia ta jest wówczas automatycznie przesuwana tak aby kursor pozostał w polu widzenia. Po przesunięciu kursora do innej linii, przesunięta linia wraca do swojej pierwotnej pozycji.

Aby przejść do początku całego pliku należy nacisnąć:

```
<CTRL> <SHIFT> H
```

### 2.4.1 Tabulatory

Przesunięcie kursora do najbliższego ogranicznika tabulacji następuje po naciśnięciu klawisza <TAB>.

Aby wcześniej ustawić taki ogranicznik, należy przesunąć kursor w żądane miejsce i nacisnąć:

<SHIFT> <SET TAB>

Skasowanie ogranicznika następuje po przesunięciu kursora na ogranicznik i naciśnięciu:

<CTRL> <SET TAB>

## 2.4.2 Przeszukiwanie tekstu

Edytor ma możliwość odnalezienia we wpisanym tekście zadanego łańcucha znakowego (1-32). Jest to bardzo pomocne przy przeskakiwaniu z jednego miejsca pliku na drugie. Należy wprowadzić:

<CTRL> <SHIFT> F

W linii komunikatów pojawi się napis:

Find?

Jeżeli komenda FIND była już używana zostanie także wyświetlony łańcuch który był poszukiwany poprzednio. Chcąc odszukać następną pojavienie się w tekście tego łańcucha należy nacisnąć <RETURN>.

W przeciwnym wypadku należy wprowadzić nowy łańcuch znakowy i wówczas nacisnąć <RETURN>.

Poszukiwanie zadanego łańcucha rozpoczyna się od miejsca w którym znajduje się kursor. Po jego odnalezieniu, kursor ustawia się na pierwszym znaku. Jeżeli łańcuch nie zostanie odnaleziony pojawi się napis:

not found

## 2.5 Poprawianie tekstu

Kolejne sześć paragrafów zawiera informacje jak poprawiać i kasować tekst znajdujący się w buforze edytora. Paragraf siódmy pokazuje jak odzyskać tekst, który został omyłkowo skasowany.

### 2.5.1 Kasowanie znaków

Aby skasować znak znajdujący się pod kursorem należy wprowadzić:

<CTRL> <DELETE>

Znaki znajdujące się po prawej stronie wymazanego znaku zostaną automatycznie przesunięte o jedno miejsce w lewo.

Aby skasować znak po lewej stronie kursora należy nacisnąć:

<BACK S>

Jeżeli aktualny był tryb „ZAMIANA” na miejsce tego, znaku zostanie wprowadzona spacja. W trybie „WSTAWIANIE” znak zostanie skasowany a wszystkie znaki po jego prawej stronie zostaną przesunięte o jedną pozycję w lewo.

### 2.5.2 Wstawianie / zamiana znaków

Jak już wspomniano, możliwe są dwa tryby wprowadzania tekstu: tryb ZAMIANA i WSTAWIANIE. Przy pierwszym wejściu do systemu ACTION! ustawia się automatycznie tryb ZAMIANA. Zmianę trybu umożliwia komenda:

<CTRL> <SHIFT> I

Działanie niektórych komend edytora zależy od trybu, który jest aktualny.

Przez naciśnięcie klawiszy <CTRL> <INSERT> można w miejsce kursora wprowadzić znak pusty (spację). Tekst znajdujący się po prawej stronie kursora zostanie przesunięty o jedno miejsce w prawo.

**UWAGA:** Jeżeli aktualny jest tryb WSTAWIANIE, wystarczy tylko nacisnąć klawisz spacji.

### 2.5.3 Kasowanie całej linii

Aby skasować linię w której znajduje się kursor należy nacisnąć

<SHIFT> <DELETE>

Wszystkie następane linie, programu zostaną przesunięte o jeden wiersz w górę.

### 2.5.4 Wstawianie linii

Aby wstawić pustą linię powyżej linii, w której znajduje się kursor, należy nacisnąć:

<SHIFT> <INSERT>

Wszystkie następane linie programu zostaną przesunięte o jeden wiersz w dół.

### 2.5.5 Dzielenie i łączenie linii

W celu podzielenia danej linii na dwie sąsiadujące ze sobą należy ustawić kursor na znak mający być pierwszym znakiem drugiej linii. Następnie należy nacisnąć:

<CTRL> <SHIFT> <RETURN>

**UWAGA:** Jeżeli aktualny jest tryb WSTAWIANIE wystarczy ustawić kursor na żądanej pozycji i nacisnąć <RETURN>.

Wszystkie następane linie tekstu zostaną przesunięte o jeden wiersz w dół.

Aby połączyć dwie sąsiadujące linie ze sobą, należy ustawić kursor na pierwszym znaku drugiej linii i nacisnąć:

<CTRL> <SHIFT> <BACK S>

### 2.5.6 Podstawianie tekstu

Edytor *ACTION!* umożliwia podstawianie całego łańcucha tekstowego w miejsce innego. Edytor pyta się najpierw o nowy łańcuch, a następnie ten, w miejsce którego należy go wstawić. Następuje poszukiwanie „starego” łańcucha (począwszy od pozycji kursora) i jego zamiana na „nowy”. Aby rozpocząć tę komendę należy wcisnąć:

<CTRL> <SHIFT> S

W linii komunikatów pojawi się napis:

Substitute?

Jeżeli komenda ta była już używana, pojawi się również łańcuch poprzednio wprowadzony. Aby go ponownie wprowadzić wystarczy nacisnąć <RETURN>. W przeciwnym wypadku należy wprowadzić nowy tekst i dopiero wówczas nacisnąć <RETURN>.

Po naciśnięciu klawisza <RETURN> w linii komunikatów pojawi się napis:

for?

Również i w tym przypadku, jeżeli komenda ta była wcześniej używana, pojawi się łańcuch który był wymazywany poprzednim razem. Można wówczas od razu nacisnąć <RETURN> lub wprowadzić nowy tekst.

Następnie edytor próbuje wykonać żadaną zamianę. Jeżeli łańcuch, który ma być wymieniony na nowy nie zostanie odnaleziony, w linii komunikatów pojawi się napis:

Not found

Jeżeli przed wykonaniem innych poprawek, w programie wykona się ponownie komendę <CTRL> <SHIFT> S, edytor wykona to samo podstawienie, po raz drugi. Pozwala to wymienić częściej niż raz występujący łańcuch na nowy bez pojawiania się każdorazowo napisów „Substitute?” i „for?”.

**UWAGA:** Używając tej komendy można skasować wybrany łańcuch tekstowy. W takim przypadku za nowy łańcuch nie należy nic wprowadzać.

## 2.5.7 Odzyskiwanie skasowanego tekstu

Edytor *ACTION!* pozwala przywrócić zmienionej linii jej poprzednią formę. Jest to potrzebne, gdy w czasie edycji popełniło się błąd. W tym celu należy nacisnąć:

<CTRL> <SHIFT> U

**UWAGA:** Komenda ta działa poprawnie tylko wtedy jeśli w między czasie nie nastąpiło przejście do innej linii.

Jeżeli przypadkowo została skasowana cała linia, można przywrócić jej; poprzednią formę przez wykonanie:

<CTRL> <SHIFT> P

Więcej informacji na ten temat znajduje się w §2.7.

**UWAGA:** Etykiety umieszczone w zmienionej lub skasowanej linii nie są odzyskiwane.

## 2.6 Okna tekstowe

Centralna część ekranu nazwana została oknem tekstowym. Kolejne pięć paragrafów opisuje komendy edytora służące do manipulowania, tworzenia i kasowania okien. Termin „okno aktualne” oznaczać będzie okno w którym znajduje się kursor.

### 2.6.1 Sterowanie oknem tekstowym

Istnieje możliwość podniesienia lub opuszczenia okna o jedną linię.

Po przesunięciu kursora poza górną granicę ekranu okno zostaje automatycznie podniesione o jeden wiersz, tak aby kursor pozostał na ekranie. Analogicznie dokonuje się przesunięcia okna o jeden wiersz w dół.

Przy dużych programach ten rodzaj przesuwania okna jest bardzo czasochłonny. Dlatego też możliwe jest przesuwanie okna o całą jego wysokość.

w górę: <CTRL> <SHIFT> <up arrow>

w dół: <CTRL> <SHIFT> <down arrow>

Dla zachowania ciągłości, przy podnoszeniu w górę, górna linia starego okna staje się dolną linią nowego, a przy opuszczaniu w dół, dolna linia starego okna będzie najwyższą linią nowego.

Edytor dopuszcza także przesuwanie okna w kierunku poziomym.

Aby przesunąć okno o jeden znak w prawo należy nacisnąć:

<CTRL> <SHIFT> ]

natomiast w lewo:

<CTRL> <SHIFT> [

### 2.6.2 Tworzenie drugiego okna

Przy pierwszym wprowadzeniu systemu *ACTION!* tworzone jest tylko jedno okno tekstowe. Można utworzyć drugie, przez naciśnięcie:

<CTRL> <SHIFT> 2

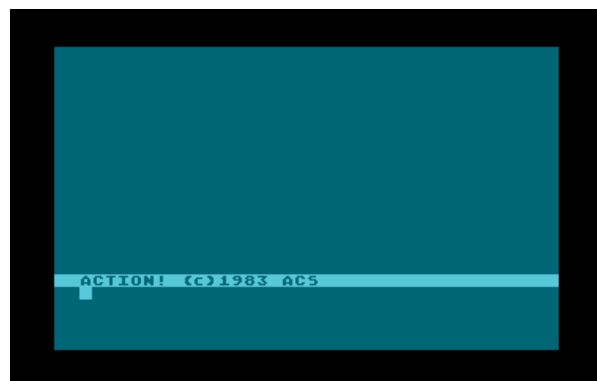
Ekran przyjmie następującą postać:



Obszar znajdujący się powyżej linii komunikatów nazywa się oknem 1, a poniżej oknem 2.

Można używać obydwóch okien niezależnie od siebie i w ten sposób pracować na dwóch różnych plikach wejściowych.

**UWAGA:** Rozmiar okna nr 1 można ustawić z poziomu monitora. Więcej informacji na ten temat znajduje się w §2.5.



### 2.6.3 Zamiana okna aktualnego

Aby przejść z okna 1 do okna 2 należy nacisnąć:

<CTRL> <SHIFT> 2

Jeżeli okno 2 nie istniało, edytor utworzy je, a następnie umieści w nim kursor.

Aby ponownie aktualne było okno nr 1 należy wprowadzić:

<CTRL> <SHIFT> 1

### 2.6.4 Kasowanie zawartości okna

Aby usunąć plik tekstowy należy umieścić kursor, w którym się znajduje i nacisnąć:

<SHIFT> <CLEAR>

W linii komunikatów pojawi się napis:

Clear?

Należy odpowiedzieć „Y” lub „N”. Jeżeli w pliku widocznym w danym oknie były dokonywane poprawki i nowa wersja nie była jeszcze zapamiętana, w linii komunikatów pojawi się napis:

Not saved, Delete?

Zadając to pytanie edytor, upewnia się czy zamiar skasowania zmienianego pliku jest świadomy, czy też zaszła pomyłka.

**UWAGA:** Komenda ta wymazuje z pamięci cały plik, a nie tylko jego część widoczną w oknie.

### 2.6.5 Kasowanie okna

Aby skasować jedno z dwóch okien, należy ustawić kursor w danym oknie i wykonać komendę:

<CTRL> <SHIFT> D

W linii komunikatów pojawi się napis:

Delete Window?

Należy odpowiedzieć „Y” lub „N”. Ponieważ kasowanie okna wiąże się jednocześnie z kasowaniem pliku, który się w nim znajduje, może dodatkowo pojawić się pytanie (zob. §2.6.4.):

Not saved, Delete?

Po skasowaniu jednego okna obszar ekranu, który ono zajmowało zostaje przypisany do drugiego z nich. Jeżeli kasowane było okno 1, okno 2 staje się oknem nr 1.

## 2.7 Przesuwanie/kopiowanie bloków tekstu

Edytor umożliwia przesuwanie lub kopiowanie bloków tekstu, wykorzystując w tym celu bufor kopii.

Kasując pojedynczy wiersz przy użyciu komendy <SHIFT> <DELETE>, wiersz ten jest automatycznie umieszczany w buforze kopii i można go następnie umieścić w dowolnym miejscu programu korzystając z komendy

<CTRL> <SHIFT> P

Bufor kopii, jest wymazywany przy każdym użyciu komendy <SHIFT> <DELETE>. Jedyny wyjątek, od tej reguły jest wtedy, gdy używa się tej. komendy kilkakrotnie bez wprowadzania między nimi innych komend lub tekstu. W takim przypadku bufor nie jest opróżniany lecz kolejne linie są dopisywane do niego. Mamy wówczas do czynienia z blokiem tekstu.

Aby więc przesunąć cały blok tekstu w inne; miejsce należy ustawić kursor” na pierwszą linię wybranego bloku i naciskać <SHIFT> <DELETE> tak długo aż zostanie on skasowany. Następnie; należy przesunąć kursor do linii powyżej której chce się dopisać znajdujący się w buforze blok tekstu i nacisnąć



<CTRL> <SHIFT> P

Kopiowanie fragmentów tekstu odbywa się w ten sam sposób, z tym że najpierw kopiowany blok tekstu należy dołączyć z powrotem w miejsce gdzie się znajdował. Nie powoduje to wymazania bufora, a więc: przesuując kursor i używając komendy <CTRL> <SHIFT> P można skopiować ten sam blok w różnych miejscach.

## 2.8 Etykiety

Etykiety pozwalają oznaczyć dowolne miejsce w tekście. Aby ustawić etykietę w miejscu gdzie znajduje się kursor należy nacisnąć:

<CTRL> <SHIFT> T

W linii komunikatów pojawi się napis:

tag id:

W odpowiedzi wprowadza się pojedynczy znak identyfikujący tą etykietę i naciska <RETURN>. Jeżeli wprowadzony znak był wcześniej użyty jako identyfikator etykiety, poprzednia etykieta (znajdująca się w innym miejscu) będzie wymazana.

Podczas dalszej edycji można przesunąć kursor do określonej etykiety wykonując komendę:

<CTRL> <SHIFT> G

W linii komunikatów pojawi się napis:

tag id:

Należy wówczas wprowadzić identyfikator etykiety do której ma być przesunięty kursor. Jeżeli etykieta taka nie istnieje, edytor wydrukuje komunikat:

tag not set

**UWAGA:** Wszystkie operacje związane ze zmianą zawartości linii (wstawianie znaków, kasowanie, zamiana, dzielenie i łączenie linii) powodują wymazanie wszystkich etykiet w tej linii.

## Rozdział 3 Porównanie edytorów ACTION! i ATARI

### 3.1 Komendy wspólne

<SHIFT>	Użyty w połączeniu z klawiszami literowymi powoduje ich zamianę z dużych na małe litery alfabetu. Dla pozostałych klawiszy powoduje wybór komendy lub znaku napisanego wyżej. <SHIFT> musi być wciśnięty podczas naciskania następnego klawisza.
<CTRL>	Używany w połączeniu z jednym lub więcej klawiszami dla komend i znaków specjalnych edytora. Podczas naciskania następnego klawisza <CTRL> musi być wciśnięty. np. <CTRL> <up arrow> symbolizuje komendę przesuującą kursor o jeden wiersz w górę.
<Atari>	Znaki wyświetlane na ekranie są przedstawiane na odwrotnym tle. Powrót do normalnej pracy następuje po powtórnym naciśnięciu tego klawisza.
<ESC>	Pozwala następujący po nim znak sterujący wprowadzić jako tekst.
<LOWR>	Powoduje, że wprowadzane będą małe litery alfabetu.
<SHIFT> <CAPS>	Powoduje, że wprowadzane są tylko duże litery alfabetu.
<SHIFT> <INSERT>	Wstawia jedną pustą linię w miejsce, gdzie znajdował się kursor. Linia z kursorem oraz wszystkie następne przesuwane są o jeden wiersz w dół.
<CTRL> <INSERT>	W miejscu gdzie znajdował się kursor wstawiany jest znak pusty. Wszystkie kolejne znaki tej linii przesuwane są o jedną pozycję w prawo.
<CTRL> <up arrow>	Przesuwa kursor o jeden wiersz w górę.
<CTRL> <down arrow>	Przesuwa kursor o jeden wiersz w dół.
<TAB>	Przesuwa kursor do następnego tabulatora, jeżeli taki istnieje. Jeżeli w tej pominiętej części wiersza nie było do tej pory tekstu, wstawiane są tam spacje.
<SHIFT> <SET TAB>	Ustawia tabulator w miejsce, gdzie znajduje się kursor.
<CTRL> <CLR TAB>	Jeżeli w pozycji zajmowanej przez kursor znajdował się tabulator, jest on kasowany.

### 3.2 Komendy różne dla obydwóch edytorów

<BREAK>	Komenda ta jest używana tylko przez edytor ATARI.
<SHIFT> <CLEAR>	Wymazuje plik znajdujący się w oknie aktualnym. Jeżeli plik ten nie został wcześniej zapamiętany, edytor informuje o tym fakcie użytkownika. Pozwala to ewentualnie unieważnić komendę.
<RETURN>	W trybie ZAMIANA przesuwa kursor do początku następnej linii. W trybie WSTAWIANIE do tekstu zostanie wstawiony <RETURN>.
<SHIFT> <DELETE>	Wymazuje linię, w której znajduje się kursor. Wszystkie następne linie są przesuwane o jeden wiersz w górę. Komenda ta może być powtarzana. Kasowane linie są zapamiętywane w buforze kopii, aby umożliwić ewentualnie kopiowanie/przesuwanie bloków tekstu. Zobacz opis komendy <CTRL> <SHIFT> P oraz §2.7.
<BACK S>	Jeżeli edytor znajduje się w trybie ZAMIANA (zob. <CTRL> <SHIFT> I) wówczas znak znajdujący się po lewej stronie kursora zastępowany jest spacją. W trybie WSTAWIANIE kasowane są wszystkie znaki po lewej stronie kursora, a pozostała część linii jest przesuwana w lewo.
<CTRL> <right arrow>	Przesuwa kursor o jeden znak w prawo. Po napotkaniu prawej granicy okna edytor utrzymuje kursor w polu widzenia poprzez przesuwanie linii w lewo.
<CTRL> <left arrow>	Przesuwa kursor o jeden znak w lewo. Jeżeli osiągnięty jest lewy koniec okna, a nie osiągnięto jeszcze początku linii, tekst jest przesuwany o jeden znak w prawo.

### 3.3 Komendy unikalne dla edytora ACTION!

<CTRL> <SHIFT> D	Kasuje okno, w którym znajduje się kursor. Jego zawartość jest wymazywana z pamięci, okno znika z ekranu i aktualne staje się okno drugie.
<CTRL> <SHIFT> F	Odnajduje w tekście wyspecyfikowany łańcuch znaków. Po jego odnalezieniu, okno i kursor zostają przesunięte w to miejsce.
<CTRL> <SHIFT> G	Odnajduje w pliku (licząc od danej pozycji) wyspecyfikowaną przez użytkownika etykietę. Po jej odnalezieniu kursor jest do niej przesuwany, a na ekranie wyświetlany jest otaczający ją tekst.
<CTRL> <SHIFT> H	Przesuwa kursor do początku pliku.
<CTRL> <SHIFT> I	Zmienia aktualny tryb pracy edytora: ZAMIANA/WSTAWIANIE. (Edytor rozpoczyna pracę od trybu WSTAWIANIE). Nowy tryb jest wyświetlany w linii komunikatów. Komenda ta ma wpływ na działanie: komend <BACK S> oraz <RETURN>.
<CTRL> <SHIFT> M	Przekazuje sterowanie na poziom monitora systemu ACTION! (zob. część III).
<CTRL> <SHIFT> P	Komenda ta powoduje dołączenie w miejscu ustawienia kursora linii znajdujących się w buforze kopii.
<CTRL> <SHIFT> R	Odczytuje plik tekstowy z urządzenia zewnętrznego. Podana nazwa pliku musi być zgodna z systemem operacyjnym DOS, którego się używa. Jeżeli w nazwie pliku nie jest wyspecyfikowane urządzenie z którego ma nastąpić odczyt, edytor automatycznie przyjmuje, że jest to D1.
<CTRL> <SHIFT> S	Zamiana łańcuchów tekstowych. Komenda ta pozwala umieścić grupę znaków (do 32) w miejsce wyspecyfikowanego łańcucha. Jeżeli stary łańcuch występuje kilkakrotnie, komendą tą można go zmienić we wszystkich miejscach tekstu, w których występuje.
<CTRL> <SHIFT> T	Ustawia etykietę w pozycji, w której znajduje się kursor. Etykieta ta nie stanowi części tekstu i jest wykorzystywana tylko do sterowania kursorem.
<CTRL> <SHIFT> U	Kasowanie dokonanych poprawek tekstu. Komenda ta przywraca zmienionym liniom ich pierwotną formę. Etykiety, które znajdowały się w tych liniach nie są pamiętane. Komenda ta ma znaczenie tylko wtedy, gdy linia nie została skasowana przez <SHIFT> <DELETE> i kursor nie był przeniesiony do innego wiersza.
<CTRL> <SHIFT> W	Wyprowadza plik na urządzenie zewnętrzne. Nazwa pliku musi być zgodna z używanym systemem operacyjnym DOS.
<CTRL> <SHIFT> ]	Przesunięcie okna o jedną kolumnę w prawo (pożyteczne podczas edycji programów z dużą ilością wcięć).
<CTRL> <SHIFT> [	Przesuwa okno o jedną kolumnę w lewo.
<CTRL> <SHIFT> <up arrow>	Przesuwa okno o całą jego wysokość w górę. Dla zachowania ciągłości, górna linia poprzedniego okna staje się dolną linią nowego.
<CTRL> <SHIFT> <down arrow>	Przesuwa okno o całą jego wysokość w dół. Dolna linia starego okna staje się górną linią nowego.
<CTRL> <SHIFT> 1	Aktualne: staje się okno pierwsze. Kursor powraca na poprzednią pozycję.
<CTRL> <SHIFT> 2	Aktualne staje się okno nr 2. Jeżeli do tej pory okno to nie istniało, edytor powoduje jego utworzenie.
<CTRL> <SHIFT> >	Przesuwa kursor do końca linii. Na ekranie będą widoczne ostatnie 38 znaki danej linii.
<CTRL> <SHIFT> <	Przesuwa kursor do początku linii. Na ekranie będą widoczne pierwsze 38 znaki linii.
<CTRL> <SHIFT> <BACK S>	Jeżeli kursor znajduje się na początku linii, w wyniku działania tej komendy, linia ta zostanie dołączona do linii ją poprzedzającej. Wszystkie linie następne są przesuwane o jeden wiersz w górę. We wszystkich innych przypadkach komenda ta działa tak samo jak <BACK S>
<CTRL> <SHIFT> <RETURN>	Wstawia <RETURN> do tekstu. Linia zawierająca kursor jest dzielona na dwie części. Część linii na lewo od kursora pozostaje linią aktualną. Drugą jej część wstawiana jest wiersz niżej.

## Rozdział 4 Uwagi techniczne

### 4.1 Pliki tworzone przez inne edytory tekstu

Edytor *ACTION!* nie może pracować na plikach nie zawierających znaków końca linii <RETURN> (EOL lub mających linie dłuższe niż 240 znaków.

Dla wygody, długość linii nie powinna być dłuższa niż maksymalna ilość znaków w wierszu na drukarce, której się używa.

### 4.2 Błąd „przepelnienia pamięci”

Sytuacja taka może wystąpić jeżeli plik na którym się pracuje jest już zbyt duży lub praca z edytorem trwa już stosunkowo długo i w tym czasie zostało wykonanych sporo podstawień i/lub kopiowań tekstu.

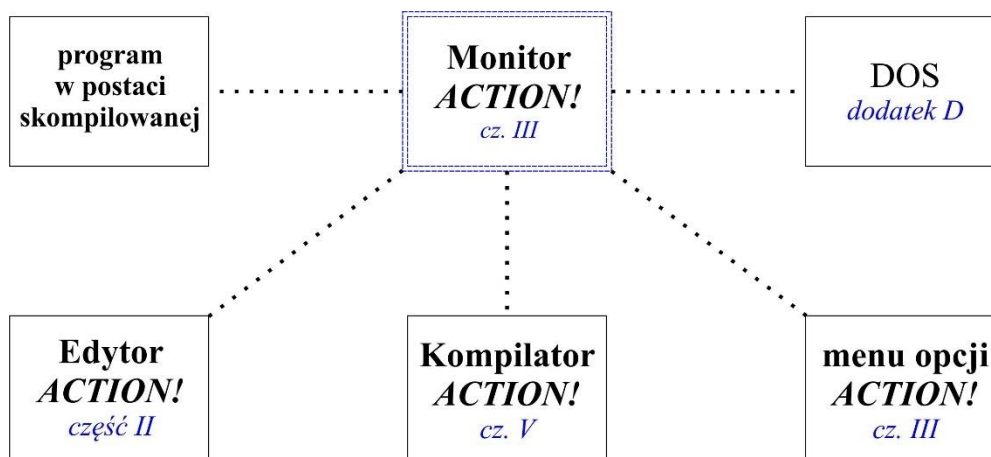
Jeżeli błąd ten wystąpi należy natychmiast zapisać dany plik w pamięci zewnętrznej, a następnie ponownie uruchomić system *ACTION!* komendą monitora „BOOT”. Można wówczas wrócić pod edytor, wczytać ponownie do pamięci komputera plik na którym się pracowało i kontynuować edycję.

## Część III      Monitor *ACTION!*

<b>Rozdział 1</b>	<b>Wstęp</b>	<b>18</b>
1.1	Używane terminy	18
1.2	Cechy i możliwości monitora <i>ACTION!</i>	18
<b>Rozdział 2</b>	<b>Komendy monitora</b>	<b>19</b>
2.1	BOOT - restart systemu	19
2.2	COMPILE - kompilacja programów	19
2.3	DOS - przejście na poziom systemu operacyjnego	19
2.4	EDIT - przejście na poziom edytora	19
2.5	OPTIONS - wybór opcji monitora	20
2.6	PROCEED - kontynuacja zatrzymanego programu	21
2.7	RUN - uruchomienie programu	21
2.8.	SET - ustawianie wartości komórek pamięci	21
2.9	WRITE - zapamiętywanie skompilowanych programów	22
2.10	EXECUTE - komendy natychmiastowego wykonania	22
2.11	? - wyświetlanie wartości komórek pamięci	22
2.12	* - „zrzucanie” pamięci	23
<b>Rozdział 3</b>	<b>Możliwości testowania programu</b>	<b>24</b>
	Opcja TRACE	24

# Rozdział 1 Wstęp

W części III został opisany monitor - centrum sterowania całego systemu *ACTION!*. Pozwala on na dostęp do wszystkich funkcji systemu.



Zgłoszenie się monitora charakteryzuje się tym, że górna linia ekranu ma odwrócone tło i po lewej stronie zawiera znak oraz kursor.

## 1.1 Używane terminy

TERMIN	GDZIE ZDEFINIOWANY
<adres>	część IV
<stałe kompilacji>	część IV
<specyfikacja pliku>	poniżej
<identyfikator>	część IV
<instrukcja>	część IV
<wartość>	część IV

Przez <specyfikację pliku> uważa się w tej części standardowy specyfikator pliku na Atari zawierający symbol urządzenia (P:, C:, D1:, D2: itd.) oraz nazwę pliku w przypadku dysku.

## 1.2 Cechy i możliwości monitora ACTION!

Monitor *ACTION!* posiada dwie ważne cechy - linię komend i obszar komunikatów.

Komunikacja z monitorem *ACTION!* następuje poprzez linię komend.

Jest to górna linia ekranu zawierająca na początku znak „>” oraz kursor.

Komendy są rozpoznawane poprzez pierwszy znak wprowadzony po znaku „>”.

Tak więc komendy „E”, „Edit” i „Ejunk” są równoważne i wszystkie oznaczają, że sterowanie ma być przekazane do poziomu edytora.

Poniżej linii komend znajduje się obszar komunikatów. Jest to duży obrysowany blok w środku ekranu. Obszar ten ma kilka zastosowań.

Po uruchomieniu programu jest on wykorzystywany do wyświetlania wyników. Może być użyty do śledzenia działania programu. Jeżeli system operacyjny lub kompilator *ACTION!* odnajdzie błąd w obszarze tym pojawi się numer błędu i tekst znajdujący się dookoła linii programu, w której ten błąd został wykryty.

## Rozdział 2 Komendy monitora

### 2.1 BOOT – restart systemu

Czasami istnieje potrzeba ponownego uruchomienia całego systemu *ACTION!* z poziomu monitora np. w przypadku jakiegoś fatalnego błędu lub powrotu z poziomu systemu operacyjnego DOS.

W tym celu należy wprowadzić „BOOT” i nacisnąć <RETURN>.

#### Przykład:

```
BOOT <RETURN>
B <RETURN>
```

**UWAGA:** Tekst znajdujący się w buforze edytora, skompilowane programy i ich zmienne zostaną skasowane.

### 2.2 COMPILE – kompilacja programów.

W systemie *ACTION!* uruchomienie programu z poziomu monitora może nastąpić dopiero po jego przetworzeniu przez kompilator. Wywołanie kompilatora następuje przez użycie komendy:

```
COMPILE „<specyfikacja pliku>„
```

<specyfikacja pliku> jest opcją, która pozwala skompilować program znajdujący się w pamięci zewnętrznej (dyskietka, kaseeta itp.).

Jeśli nie jest to wyspecyfikowane, kompilowana jest zawartość bufora edytora. Jeżeli używa się dwóch okien kompilowany jest plik w którym znajdował się kursor podczas wyjścia z poziomu edytora.

Jeżeli kompilator wykryje błąd składni podczas kompilacji programu w obszarze komunikatów monitora pojawi się numer błędu oraz linia w której ten błąd się znajduje. Sterowanie powraca na poziom monitora.

#### Przykład:

COMPILE <RETURN>	rozkaz skompilowania programu z aktualnego okna edytora
C <RETURN>	
C „C:” <RETURN>	kompilacja programu z kasety
C „D1:PRIME.ACT” <RETURN>	kompilacja programu PRIME.ACT z mechanizmu dyskowego nr 1
COMPILE „PRIME.ACT” <RETURN>	

Zauważcie, że w ostatnim przykładzie, nie jest podana nazwa urządzenia. W takim przypadku przyjmowane jest automatycznie urządzenie D1.

### 2.3 DOS – przejście na poziom systemu operacyjnego

Przeniesienie pracy na poziom OS/A+,DOS XL lub Atari DOS następuje po wprowadzeniu „DOS” i naciśnięciu <RETURN>.

#### Przykład:

```
DOS <RETURN>
D <RETURN>
```

**UWAGA:** Ponieważ Atari DOS używa tego samego obszaru pamięci co *ACTION!* powinno się wcześniej zapamiętać wszystkie pliki.

### 2.4 EDIT – przejście na poziom edytora

Aby tego dokonać należy wprowadzić „EDITOR” i nacisnąć <RETURN>.

#### Przykład:

```
EDITOR <RETURN>
E <RETURN>
```

Menu monitora pozwala zmienić niektóre parametry działania systemu. Aby dostać się do menu należy wprowadzić „OPTIONS” i nacisnąć <RETURN>.

**Przykład:**

```
OPTIONS <RETURN>
```

```
O <RETURN>
```

Poszczególne opcje są wyświetlane: w linii komend. Jeżeli daną opcję chce się zmienić należy wprowadzić żadaną wartość i nacisnąć <RETURN>. Jeżeli chce się zakończyć proces zmiany opcji systemu należy nacisnąć <ESC>.

## 2.5 OPTIONS – wybór opcji monitora

Poniżej podana jest pełna lista opcji. Dla każdej z nich pokazana jest postać linii komend, stan początkowy oraz części systemu *ACTION!* na które ma wpływ (M – monitor, C – kompilator, E – edytor).

<b>Display?</b>	Y	M, C, E
-----------------	---	---------

Zwiększa prędkość kompilacji i operacji dyskowych wejścia/wyjścia przez wyłączenie ekranu. Aby to uzyskać należy wprowadzić „N”.

<b>Bell?</b>	Y	M, C, E
--------------	---	---------

Przy wykryciu błędu przez monitor, kompilator lub edytor wysyłany jest sygnał dźwiękowy. Również wywołanie monitora jest sygnalizowane w ten sam sposób. Wprowadzenie „N” spowoduje zablokowanie tego sygnału.

<b>Case sensitive?</b>	N	C
------------------------	---	---

Jeżeli opcja ta ma wpisane „Y”, zmienne napisane innym rodzajem liter (małe lub duże litery alfabetu) są traktowane jako różne, np. „count” różni się od „Count” i „COUNT”. Instrukcje języka takie jak FOR, WHILE, DO muszą być napisane dużymi literami. Dla wygody programisty po wprowadzeniu systemu *ACTION!* opcja ta jest ustawiona na „N” co oznacza, że niezależnie jakimi literami są napisane zmienne, są traktowane jako te same.

<b>Trace?</b>	N	C
---------------	---	---

Dzięki tej opcji można śledzić proces kompilacji programu. Jeżeli jest ona ustawiona na „Y”, kompilator umieszcza w obszarze komunikatów monitora każde wywołanie procedur razem z parametrami z jakimi je wywołał. W rozdziale 4 znajduje się więcej informacji na ten temat.

<b>List?</b>	N	C
--------------	---	---

Kompilator może wyświetlać linię, która jest aktualnie kompilowana. Należy w tym celu wprowadzić „Y”.

<b>Window 1 size?</b>	18	E
-----------------------	----	---

Rozmiar okna nr 1 edytora jest ustawiany wprost przez użytkownika. Rozmiar okna nr 2 jest ustawiany pośrednio przez rozmiar okna nr 1. Łącznie obydwa okna zajmują 23 linie. Jeżeli istnieją dwa okna, każde z nich nie może zajmować mniej niż 5 linii i więcej niż 18.

Jeżeli wprowadzi się liczbę większą niż 18, komputer przyjmuje, że rozmiar okna wynosi 18. Podobnie, jeżeli wprowadzi się liczbę mniejszą niż 5, rozmiar okna będzie wynosił 5 wierszy.

<b>Line size?</b>	120	E
-------------------	-----	---

Długość linii jest liczbą znaków w danym wierszu licząc od lewego marginesu. Opcja ta umożliwia sterowanie długością linii listowanych na drukarce. Edytor wysyła sygnał dźwiękowy jeśli zadana długość jest przekroczona.

**UWAGA:** Można ustawić długość linii większą niż 240 znaków. Monitor nie wykrywa tego błędu. Linie takie będą skracane przez edytor do dopuszczalnych 240 znaków.

<b>Left margin?</b>	2	M, E
---------------------	---	------

Opcja ta ma na celu pełne wykorzystanie ekranów, które mają możliwość wyświetlania najbardziej skrajnych dwóch znaków (nie wszystkie monitory TV mają tę możliwość). Sugeruje się aby ustawiać lewy margines jak najbliżej krawędzi ekranu, jednakże nie na tyle blisko aby miało to przeszkadzać w pracy. Normalnie jest on ustawiony na 2 jednak użytkownik ma możliwość wprowadzenia liczby z zakresu od 0 do 39. Po wprowadzeniu określonej liczby należy nacisnąć <RETURN>.



<b>EOL character?</b>	(puste)	E
-----------------------	---------	---

Znak EOL (koniec linii) jest znakiem, który edytor *ACTION!* wyświetla na końcu linii. Normalnie jest to znak pusty, lecz użytkownik ma możliwość wprowadzenia dowolnego, innego znaku, który będzie widoczny na ekranie.

## 2.6 PROCEED – kontynuacja zatrzymanego programu

Program, którego wykonanie została zatrzymane klawiszem <BREAK> używającym bibliotecznej procedury „Break”, może być kontynuowany od tego samego miejsca. Należy w tym celu wprowadzić „PROCEED” i nacisnąć <RETURN>.

### Przykład:

```
PROCEED <RETURN>
P <RETURN>
```

## 2.7 RUN – uruchomienie programu

Komendą RUN można uruchomić program, który został wcześniej skompilowany i nadal pozostaje w pamięci komputera. Komenda ta ma następujący format:

```
RUN
RUN „<Specyfikacja pliku>„
RUN <adres>
RUN <podprogram>
```

gdzie <podprogram> jest istniejącym identyfikatorem PROC lub FUNC (na przykład: dla „PROC Prime ()” można użyć „Prime” jako identyfikator procedury).

Pierwszy format jest używany do uruchomienia skompilowanego programu znajdującego się w buforze edytora.

Drugi z nich pozwala uruchomić programy zapamiętane w urządzeniach zewnętrznych. Jeżeli jest to program w postaci źródłowej (w języku *ACTION!*) jest on najpierw kompilowany przez kompilator i dopiero wtedy uruchamiany. Jeżeli program jest już w postaci skompilowanej jest on natychmiast uruchamiany.

Trzeci z formatów pozwala uruchomić program (lub podprogram), który zaczyna się pod danym adresem. Jest to użyteczne do testowania programów, które wywołują podprogramy napisane w języku maszynowym.

Format czwarty jest używany do uruchomienia pojedynczej procedury lub funkcji programu, który został wcześniej skompilowany.

Po wykonaniu programu sterowanie powraca na poziom monitora. Jeżeli wystąpiły jakieś znaczące błędy powrót na poziom monitora następuje po naciśnięciu <SYSTEM> <RESET>.

### Przykład:

RUN <RETURN>	uruchomienie skompilowanego programu z bufora edytora
R <RETURN>	
RUN „C:” <RETURN>	ściągnięcie programu z kasety, skompilowanie go, a następnie uruchomienie
RUN „PRIME.ACT” <RETURN>	ściągnięcie programu PRIME.ACT
R „D1:PRIME.ACT” <RETURN>	z dyskietki nr 1, skompilowanie i uruchomienie
R \$400 <RETURN>	uruchomienie programu od adresu \$400
RUN 1024 <RETURN>	uruchomienie programu od adresu 1024
R Prime <RETURN>	uruchomienie wcześniej skompilowanej procedury „Prime ()”
RUN PrintE() <RETURN>	uruchomienie funkcji bibliotecznej do wydrukowania na ekranie łańcucha tekstu

## 2.8 SET – ustawianie wartości komórek pamięci

Komenda monitora SET jest identyczna jak rozkaz SET języka *ACTION!*

Zob. część IV §7.3.

## 2.9 WRITE – zapamiętywanie skompilowanych programów

Komenda ta umożliwi zapisanie skompilowanego programu (można używać nazwy plik binarny) na dyskietce. Może on być następnie uruchomiony spod systemu operacyjnego DOS. Format tej komendy wygląda następująco:

```
WRITE „<specyfikacja pliku>„ <RETURN>
```

Plik binarny z pamięci zostanie w wyniku tego zapamiętany w określonym pliku na dyskietce. Jeżeli na dyskietce ten plik do tej pory nie istniał, zostanie on utworzony. Jeżeli pole przeznaczone na ten plik jest niewystarczające lub dyskietka jest w tym miejscu uszkodzona, pojawi się komunikat o błędzie i wówczas należy podjąć próbę zapamiętania pliku ponownie.

### Przykład:

WRITE „PRIME.BIN” <RETURN>	skompilowana wersja programu PRIME zostanie zapamiętana na
W „D1:PRIME.BIN” <RETURN>	dyskietce nr 1
W „C:”	skompilowany program zostanie zapisany na kasecie

Program zapamiętany za pomocą komendy „W” może być wykonany z poziomu OS lub DOS. Zob. dodatek D.

## 2.10 XECUTE – komendy natychmiastowego wykonania

Istnieje możliwość wykonania rozkazów języka *ACTION!* lub dyrektyw kompilatora (z wyjątkiem *MODULE* i *SET*) bezpośrednio z poziomu monitora. Rozkazy takie należy poprzedzić komendą *XECUTE*, a następnie nacisnąć <RETURN>.

### Przykład:

```
XECUTE PrintE („Hello World”) <RETURN>
X trace = 255 <RETURN>
```

## 2.11 ? – wyświetlanie wartości komórek pamięci

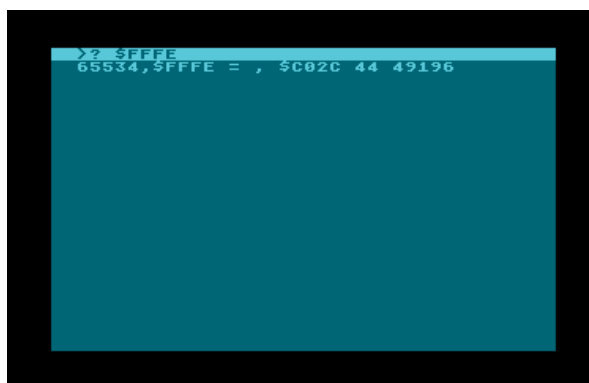
Istnieje możliwość wyprowadzenia na ekran wartości zmiennej lub konkretnej komórki pamięci. Należy w tym celu wprowadzić „?” następnie stałą, kompilacji i nacisnąć <RETURN> Format tej instrukcji jest następujący:

```
? <stała kompilacji> <RETURN>
```

Monitor *ACTION!* wyświetli na ekranie adres tej komórki pamięci (zarówno w układzie dziesiętnym jak i heksadecymalnym), a następnie znak w kodzie ATASCII odpowiadający wartości tej komórki, jej wartość w układzie heksadecymalnym i dziesiętnym (typ *BYTE*) i wartość dziesiętną typu *CARD* tej i następnej komórki łącznie. Jeżeli identyfikator użyty w tej komendzie nie jest symbolem kompilatora *ACTION!*, pojawi się komunikat:

```
„variable not declared error”
```

### Przykład:



**UWAGA:** Wynik może być inny niż się oczekiwało, ponieważ w czasie kompilacji pamięć jest zmieniana. Zob. tablicę symboli w części V.

## 2.12 \* – „zrzucanie” pamięci

Istnieje możliwość wyświetlenia zawartości kolejnych komórek pamięci począwszy od zadanego adresu. Na ekranie wydruk przyjmuje postać identyczną jak dla komendy „?” Format tej komendy jest następujący:

```
* <adres> <RETURN>
```

Przerwanie drukowania zawartości kolejnych komórek pamięci następuje po naciśnięciu klawisza <Space bar>. Chwilowe wstrzymanie listingu nastąpi po naciśnięciu <CTRL> 1. Ponowne naciśnięcie tych klawiszy powoduje kontynuację listingu.

```
>* $600
1536, $0600 ||| ♡ $0000 0 0
1537, $0601 ||| ♡ $0000 0 0
1538, $0602 ||| ♡ $0000 0 0
1539, $0603 ||| ♡ $0000 0 0
1540, $0604 ||| ♡ $0000 0 0
1541, $0605 ||| ♡ $0000 0 0
1542, $0606 ||| ♡ $0000 0 0
1543, $0607 ||| ♡ $0000 0 0
1544, $0608 ||| ♡ $0000 0 0
1545, $0609 ||| ♡ $0000 0 0
1546, $060A ||| ♡ $0000 0 0
1547, $060B ||| ♡ $0000 0 0
1548, $060C ||| ♡ $0000 0 0
1549, $060D ||| ♡ $0000 0 0
1550, $060E ||| ♡ $0000 0 0
1551, $060F ||| ♡ $0000 0 0
1552, $0610 ||| ♡ $0000 0 0
1553, $0611 ||| ♡ $0000 0 0
```

## Rozdział 3                      Możliwości testowania programu

Często zdarza się, że napisany program nie działa poprawnie.

Niektóre opcje monitora *ACTION!* pozwalają testować program krok po kroku, aż do wykrycia błędu.

### Opcja TRACE

Jeżeli opcja ta jest aktywna („Y”), można śledzić proces wykonywania programu. Na ekranie wyświetlane są wszystkie wywołania podprogramów wraz z użytymi parametrami. Dzięki temu użytkownik może wykryć błąd przez kontrolę kolejności wywołań poszczególnych procedur i funkcji oraz użytych parametrów.

Często nie jest to wystarczające do odnalezienia błędu i należy podjąć inne kroki. Jednym z nich jest możliwość zatrzymania programu w czasie jego wykonywania. Są dwa sposoby aby tego dokonać: klawisz <BREAK> i standardowa procedura biblioteczna „Break”.

#### — klawisz <BREAK>

Na poziomie edytora klawisz <BREAK> jest zablokowany. Podczas wykonywania programu można go używać tylko w niektórych przypadkach. Program zostanie zatrzymany jeżeli:

1. wykonywane są operacje wejścia/wyjścia
2. wywoływany jest podprogram z więcej niż 3 parametrami

Może to się wydawać dziwne, ale są powody do tego. System *ACTION!* nie sprawdza podczas wykonywania programu czy klawisz <BREAK> został użyty. W obydwóch wyżej opisanych przypadkach następuje odwołanie do CIO i to właśnie CIO sprawdza czy klawisz ten nie jest wciśnięty.

#### — procedura biblioteczna PROC Break ()

Jeżeli program ma być zatrzymany w innym miejscu niż opisane powyżej, należy wywołać w tym punkcie procedurę „Break”. Stosowanie tej metody jest bardziej naturalne niż używanie klawisza <BREAK>, ponieważ wiadomo wówczas dokładnie w którym miejscu został zatrzymany program.

Po zatrzymaniu programu można użyć komend monitora „+” i „?” dla sprawdzenia wartości zmiennych, których się używa.

Zamiast metod testowania programu opisanych powyżej można używać kontrolnych instrukcji „Print” drukujących wartości określonych zmiennych podczas wykonywania programu.

# Część IV Język programowania ACTION!

<b>Rozdział 1</b>	<b>Wstęp</b>	<b>27</b>
<hr/>		
<b>Rozdział 2</b>	<b>Słowa kluczowe</b>	<b>28</b>
2.1	Terminy specjalne	28
<hr/>		
<b>Rozdział 3</b>	<b>Podstawowe typy danych</b>	<b>29</b>
3.1	Zmienne	29
5.2	Stałe	29
3.3	Podstawowe typy danych	30
3.3.1	BYTE	30
3.3.2	CARDinal	30
3.3.3	INTeger	30
3.4	Deklaracje	30
3.4.1	Deklaracje zmiennych	30
3.4.2	Stałe numeryczne	31
<hr/>		
<b>Rozdział 4</b>	<b>Wyrażenia</b>	<b>32</b>
4.1	Operatory	32
4.1.1	Operatory arytmetyczne	32
4.1.2	Operatory działające na bitach	32
4.1.3	Operatory logiczne	33
4.1.4	Priorytety operatorów	34
4.2	Wyrażenia arytmetyczne	34
4.3	Proste wyrażenia logiczne	35
4.4	Złożone wyrażenia logiczne	35
<hr/>		
<b>Rozdział 5</b>	<b>Instrukcje</b>	<b>37</b>
5.1	Instrukcje proste	37
5.1.1	Instrukcje przypisania	37
5.2	Instrukcje strukturalne	38
5.2.1	Instrukcje warunkowe	38
5.2.1.1	Wyrażenia warunkowe	38
5.2.1.2	Instrukcja IF	38
5.2.2	Instrukcja pusta	38
5.2.3	Pętle	40
5.2.3.1	DO i OD	40
5.2.3.2	Instrukcja EXIT	41
5.2.4	Sterowanie pętla	42
5.2.4.1	Instrukcja FOR	42
5.2.4.2	Instrukcja WHILE	44
5.2.4.3	Instrukcja UNTIL	46
5.2.5	Zagnieżdżanie instrukcji strukturalnych	46
<hr/>		
<b>Rozdział 6</b>	<b>Procedury i funkcje</b>	<b>48</b>
6.1	Procedury	48
6.1.1	Deklaracja procedury	48
6.1.2	RETURN	49
6.1.3	Wywołanie procedury	50
6.2	Funkcje	50
6.2.1	Deklaracja funkcji	50
6.2.2	RETURN	51
6.2.3	Wywołanie funkcji	52
6.3	Zakres zmiennych	55

6.4	Parametry	55
6.5	MODULE	57
<b>Rozdział 7 Dyrektywy kompilatora</b>		<b>58</b>
7.1	DEFINE	58
7.2	INCLUDE	58
7.3	SET	59
<b>Rozdział 8 Złożone typy danych</b>		<b>60</b>
8.1	Zmienne wskaźnikowe	60
8.1.1	Deklaracja zmiennych wskaźnikowych	60
3.1.2	Operacje na zmiennych wskaźnikowych	60
8.2	Tablice	61
8.2.1	Deklaracja tablicy	62
8.2.2	Reprezentacja wewnętrzna tablic	62
8.2.3.	Korzystanie z tablic	62
8.3	Rekordy	65
8.3.1	Deklaracja rekordów	65
8.3.1.1	Deklaracja TYPE	65
5.3.1.2	Deklaracja zmiennych rekordowych	65
8.3.2	Sposób użycia rekordów	66
8.4	Dodatkowe rady odnośnie stosowania złożonych typów danych	66
<b>Rozdział 9 Dodatkowe możliwości programowania</b>		<b>72</b>
9.1	Bloki kodowe	72
9.2	Zmienne adresujące	72
9.3	Podprogramy adresujące	73
9.4	Asembler a <i>ACTION!</i>	73
9.5	Dodatkowe rady odnośnie korzystania z parametrów	74

# Rozdział 1                      Wstęp

Język programowania *ACTION!* jest sercem całego systemu. Łączy on wszystkie dobre cechy języków C i PASCAL oraz dodatkowo jest najszybszym językiem wysokiego poziomu dostępnym na komputery Atari.

W porównaniu z BASIC-em *ACTION!* jest językiem strukturalnym, a jest równie łatwym do nauczenia.

Programy napisane w języku *ACTION!* charakteryzują się dużą przejrzystością (nie ma w nich przede wszystkim tysięcy instrukcji GOTO i niezdeklarowanych zmiennych).

Struktura programu jest prosta. Jest on zbudowany z szeregu podprogramów zwanych procedurami i funkcjami. Pozwala to pisać program częściami, koncentrując się w danej chwili na jednej z nich. Później proste jest już ich połączenie. Jest to bardzo podobne np. do listy obowiązków domowych takich jak:

1. pościel łóżko
2. posprzątaj swój pokój
3. podlej kwiaty
4. ugotuj obiad

Jedyną różnicą jest to, że komputer musi wykonywać poszczególne czynności w kolejności jakiej są podane, a nie w takiej jak mu jest wygodnie.

## Rozdział 2 Słowa kluczowe

„Słowem kluczowym” będziemy nazywali wyraz lub symbol, który jest rozpoznawany przez kompilator jako część języka *ACTION!* i nie może być wykorzystywany w inny sposób niż jest to zdefiniowane w opisie języka (nie można używać tych słów jako identyfikatory).

AND	FI	OR	UNTIL	=	(
ARRAY	FOR	POINTER	WHILE	<>	)
BYTE	FUNC	PROC	XOR	#	.
CARL	IF	RETURN	+	>	[
CHAR	INCLUDE	RSH	-	>=	]
DEFINE	INT	SET	*	<	,
DO	LSH	STEP	/	<=	>
ELSE	MOD	THEN	&	\$	;
ELSEIF	MODULE	TO	%	^	"
EXIT	OD	TYPE	!	@	

### 2.1 Terminy specjalne

Zanim przejdziemy do omówienia instrukcji języka, w paragrafie tym opiszemy znaczenie niektórych terminów używanych dalej.

<b>Adres</b>	oznacza miejsce pamięci komputera. Rozkazując komputerowi umieścić określoną wartość w pamięci należy podać adres podobnie jak to się robi na pocztce. Oczywiście adres ten nie zawiera nazwy ulicy, miasta lecz tylko liczbę.
<b>Znak alfabetu</b>	dowolna litera alfabetu (mała lub duża). „Znak alfanumeryczny” dopuszcza dodatkowo cyfry 0-9.
<b>Identyfikator</b>	nazwy nadane zmiennym, procedurom itp. będą nazywane identyfikatorami. Muszą być spełnione następujące warunki: <ol style="list-style-type: none"> <li>1. nazwa musi zaczynać się od znaku alfabetu</li> <li>2. pozostałe znaki nazwy muszą być znakami alfanumerycznymi lub znakiem”-”</li> <li>3. nazwa nie może być słowem kluczowym języka</li> </ol>
<b>MSB, LSB</b>	MSB oznacza „bardziej znaczący bajt” LSB oznacza „bajt mniej znaczący”  W systemie dziesiętnym mamy do czynienia z cyframi znaczącymi, a nie bajtami np. najbardziej znaczącą cyfrą liczby 54 jest 5, a najmniej znaczącą 4.  Aby programować w języku <i>ACTION!</i> nie jest konieczna znajomość budowy wewnętrznej komputera. Należy pamiętać tylko, że liczby dwubajtowe są zapamiętywane i używana przez <i>ACTION!</i> w kolejności LSB, MSB. Znak dolara użyty przed liczbą oznacza, że jest ona zapisana w układzie heksadecymalnym (o podstawie 16)  <b>Przykład:</b> \$24FC \$0D \$88 \$F000
<b>„ ; ”</b>	średnik jest symbolem komentarza i każda linia po nim jest ignorowana przez kompilator  <b>Przykład:</b> ; to jest komentarz To nie jest komentarz i spowoduje błąd kompilacji ; ten komentarz posiada ; średnik wewnątrz var=3 ; komentarz może być ; zapisany po instrukcji ; to są 3 linie komentarza ; ; jedna z nich jest pusta
<b>&lt;abcd&gt;</b>	napis znajdujący się między tymi dwoma symbolami definiuje pewną część formatu np. <identyfikator> oznacza, że powinien być w tym miejscu wprowadzony identyfikator.
<b>{ }</b>	Wszystko co jest pomiędzy tymi nawiasami jest opcjonalne w konstrukcji formatu np. {<identyfikator>} oznacza, że w danym miejscu może być użyty identyfikator ale nie jest to konieczne.
<b> : : </b>	podobnie jak w muzyce symbole te oznaczają powtarzanie. Wszystko co znajduje się między nimi może być powtarzane wielokrotnie (łącznie z 0) np.  : <identyfikator> :  oznacza, że można w tym miejscu umieścić listę zero lub więcej identyfikatorów.
<b> </b>	symbol ten opisuje możliwość wyboru „albo” np. <identyfikator>  <adres> oznacza, że można użyć jednego albo drugiego, ale nie obydwóch jednocześnie



## Rozdział 3 Podstawowe typy danych

Zanim przejdziemy do omówienia podstawowych typów danych, omówimy wcześniej pojęcia zmiennych i stałych, które są podstawowymi obiektami na których operuje komputer.

### 3.1 Zmienne

Nazwa zmiennej musi „być identyfikatorem. Więcej informacji o zakresie działania zmiennych znajduje się w §6.3.

### 3.2 Stałe

W *ACTION!* są trzy typy stałych: stałe numeryczne, stałe tekstowe oraz stałe kompilacji.

Stałe numeryczne mogą być wprowadzane w trzech różnych formatach:

- heksadecymalnie
- dziesiętnie
- znakowo

Stałe heksadecymalne są poprzedzone znakiem dolara.

**Przykład:**

```
$4A00
$6D
$300
```

Stałe dziesiętne nie wymagają żadnego dodatkowego znaku.-

**Przykład:**

```
65500
2
324
```

**UWAGA:** Zarówno stałe heksadecymalne jak i dziesiętne: mogą mieć na początku znak „-”.

```
-$8C
-4360
```

Stałe znakowe są poprzedzone apostrofem. Są to stałe numeryczne ponieważ w maszynie są pamiętane w: postaci liczby jednobajtowej będącej kodem ATASCII odpowiadającym danemu znakowi.

**Przykład:**

```
„A
””
”5V
```

Stałe testowe zawierają łańcuch zero lub więcej znaków ujętych w cudzysłowy. W pamięci maszyny są poprzedzone liczbą określającą ich długość. Jeśli cudzysłów ma być częścią łańcucha, w miejscu gdzie ma on wystąpić należy go napisać dwa razy.

**Przykład:**

```
„to jest stała tekstowa”
„dwa ”” znajdują się w tym łańcuchu”
„58395”
„q” (stała tekstowa zawierająca jeden znak)
```

Stałe kompilacji różnią się od tych obydwóch typów stałych tym, że są używane w czasie kompilacji do ustawiania atrybutów zmiennych, procedur, funkcji oraz bloków kodów i nie są obliczane w czasie wykonywania programu. Poprawne, są następujące formaty:

- stała numeryczna
- wcześniej zdefiniowany identyfikator
- zmienna wskaźnikowa (zobacz §8.1.2.)
- dowolna suma dwóch z nich

Pierwszy format został już omówiony lecz pozostałe wymagają wytłumaczenia. Jeżeli użyje się identyfikatora, który jest już zdefiniowany (tzn. nazwy zmiennej, procedury, lub funkcji) jako stałej kompilacji użyta wartość jest adresem tego identyfikatora.

Trzeci format pozwala używać zmiennych wskaźnikowych jako stałych kompilacji. Ostatni z nich dopuszcza sumę dwóch stałych z pozostałych trzech typów.

**Przykład:**

```
cad    ; używa adresu zmiennej „cad”
$3D00 ; stała heksadecymalna
dog    ; zmienna wskaźnikowa
5+ptr  ; 5 plus zawartość wskaźnika „ptr”
$80+p  ; $80 plus adres zmiennej „p”
```

### 3.3 Podstawowe typy danych

*ACTION!* dostarcza trzech podstawowych typów danych oraz kilka ich rozszerzeń (zob. rozdział 8).

Wszystkie trzy są typami numerycznymi co pozwala na używanie formatu stałych numerycznych przy ich wprowadzaniu.

#### 3.3.1 BYTE

Typ BYTE jest używany dla dodatnich liczb całkowitych mniejszych niż 256. W maszynie są one reprezentowane przez pojedynczy bajt.

Wydawać się może, że wykorzystanie tego typu jest niewielkie. W rzeczywistości ma on dwie pozytywne cechy. Jeżeli jest użyty jako licznik pętli (WHILE, UNTIL, FOR) program wykonuje się dużo szybciej ze względu na to, że operacje na pojedynczym bajcie są znacznie prostsze.

Po drugie, ponieważ znaki są w komputerze reprezentowane przez jeden bajt, typ BYTE można wymiennie stosować jako typ znakowy CHAR.

#### 3.3.2 CARDinal

Typ CARD jest podobny do typu BYTE z tą różnicą, że w komputerze zajmuje dwa bajty. Pozwoliło to rozszerzyć jego zakres od 0 do 65535

UWAGA: CARD jest zapamiętywany w kolejności: LSB, MSB.

#### 3.3.3 INTeger

Typ INT dopuszcza liczby całkowite ujemne i dodatnie z zakresu -32768 do 32767. W maszynie jest reprezentowany przez dwa bajty

### 3.4 Deklaracje

Deklaracje są używane do definiowania obiektów, na których będzie operował program. Np. jeżeli chce się użyć zmiennej „cost” typu CARD należy to w jakiś sposób przekazać do komputera. W przeciwnym przypadku komputer nie będzie wiedział co oznacza pojawienie się w programie „cost”.

Każdy identyfikator zanim zostanie użyty musi być wcześniej zdefiniowany czy jest to nazwa zmiennej, procedury czy też funkcji.

Deklaracje procedur i funkcji będą wytłumaczone w rozdziale 6.

#### 3.4.1 Deklaracje zmiennych

```
<Typ> <identyfikator> {=<wart. pocz>} |:,<identyfikator> {=<wart. pocz.>}:|
```

gdzie :

<typ> jest typem deklarowanych zmiennych  
 <identyfikator> jest identyfikatorem nazywającym daną zmienną  
 <wart. pocz.> pozwala nadać zmiennej wartość początkową lub zdefiniować jej lokalizację w pamięci komputera

„<wart. pocz.>”, ma następującą formę:

<adres> [<wartość>]

gdzie

<adres> jest adresem zmiennej i musi to być stała kompilacji  
 <wartość> jest początkową wartością nadaną zmiennej i musi to być stała numeryczna

Należy wrócić uwagę, że można jednocześnie zadeklarować więcej niż jedną zmienną danego typu. Opcjonalne są: możliwość nakazania kompilatorowi gdzie ma umieścić daną zmienną oraz możliwość nadania danej zmiennej wartości startowej. Przykłady poniżej powinny wyjaśnić wszystkie wątpliwości.

```
BYTE top, hat ; deklaracja zmiennych top i hat typu BYTE
INT num=[0] ; deklaracja num jako zmiennej typu INT i nadanie jej wartości 0
BYTE x=$8000, ; deklaracja x jako zmiennej typu BYTE i umieszczenie jej pod adresem $8000
    y=[0] ; deklaracja i inicjacja zmiennej y
CARD ctr=[$83D4], ; deklaracja i inicjacja trzech zmiennych jako zmienne typu CARD
    bignum=[0],
    cad=[30000]
```

W ostatnich dwóch przykładach widać, że zmienne nie muszą znajdować się w tej samej linii. Kompilator *ACTION!* czyta zmienne danego typu tak długo jak znajdują się pomiędzy nimi przecinki. Należy uważać by nie postawić przecinka po ostatniej zmiennej w liście zmiennych danego typu.

Deklaracje zmiennych muszą następować natychmiast po instrukcji MODULE (zob. §7.4) lub na początku procedur lub funkcji (zob. §6.1.1 i §6.2.1). Jeżeli umieści się je w innych miejscach zostanie zasygnalizowany błąd.

### 3.4.2 Stałe numeryczne

Stałe numeryczne nie są deklarowane. Sposób w jaki są zapisane w programie definiuje ich typ. Stałe numeryczne mniejsze niż 256 są typu BYTE, a większe typu CARD. Stałe ujemne są traktowane jako typu INT.

Stale	typ
543	CARD
\$0D	BYTE
\$F42	CARD
„W	BYTE

## Rozdział 4 Wyrażenia

Wyrażenia są konstrukcjami, które mają wyznaczyć wartość zmiennych, stałych i warunków połączonych odpowiednimi operatorami. Np. „4+3” jest wyrażeniem, którego wartość wynosi 7. Jeżeli operator „+” zmienimy na „\*” wartością tego wyrażenia będzie 12 (4\*3=12).

W *ACTION!* istnieją dwa typy warażeń: arytmetyczne i logiczne. Przykład powyżej był wyrażeniem arytmetycznym. Wyrażenie logiczne daje w wyniku „true” lub „false”. **5>=7** jest fałszem ponieważ „>=”, oznacza większe lub równe. Ten typ warażeń jest używany w instrukcjach warunkowych.

### 4.1 Operatory

*ACTION!* dostarcza trzech rodzajów operatorów:

1. operatory arytmetyczne
2. operatory działające na bitach
3. operatory logiczne

Pierwsze i ostatnie są powszechnie znane. Druga klasa operatorów wykonuje działania arytmetyczne i adresowe na poziomie bitów.

#### 4.1.1 Operatory arytmetyczne

Operatory arytmetyczne są identyczne jak te używane w matematyce. Niektóre z nich zostały zmodyfikowane tak aby można je było wprowadzić z klawiatury komputera. Poniżej zamieszczona jest pełna ich lista:

- minus, np. -5
- \* mnożenie, np. 4\*3
- / dzielenie całkowite, np. 13/5 = 2
- MOD** reszta z dzielenia całkowitego, np. 13MOD5 = 3
- + dodawanie, np. 4+3
- odejmowanie, np. 4-3

Należy zwrócić uwagę, że znak „=”, nie jest operatorem arytmetycznym.

Jest on tylko używany w wyrażeniach logicznych, niektórych deklaracjach i instrukcjach przypisania.

#### 4.1.2 Operatory działające na bitach

Operatory te działają na liczbach w ich binarnej postaci. Można dzięki temu wykonywać działania podobne do tych jakie wykonuje komputer.

Oto pełna lista tych operatorów:

- & and
- % or
- ! Exclusive OR
- XOR Exclusive OR
- LSR przesuwanie bitów w lewo
- RSH przesuwanie bitów w prawo
- @ adres

Pierwsze trzy porównują liczby bit po bicie, a wynik zależy od użytego operatora. Np.:

Operator &	Bit A	Bit B	Wynik
	1	1	1
	0	1	0
	0	0	0
	1	0	0

**5 & 39**

**&**

0	0	0	0	0	0	1	0	1	<b>5</b>
0	0	1	0	0	1	1	1	<b>39</b>	
0	0	0	0	0	1	0	1	<b>5</b>	

Operator %	Bit A	Bit B	Wynik
	1	1	1
	0	1	1
	0	0	0
	1	0	1

5 % 39

0	0	0	0	0	1	0	1	5
0	0	1	0	0	1	1	1	39
<hr/>								
0	0	1	0	0	1	1	1	39

Operator !	Bit A	Bit B	Wynik
	1	1	0
	0	1	1
	0	0	0
	1	0	1

5 ! 39

0	0	0	0	0	1	0	1	5
0	0	1	0	0	1	1	1	39
<hr/>								
0	0	1	0	0	0	1	0	34

LSH i RSH przesuwają bity o jedno miejsce. Jeżeli działająca typach dwubajtowych (CARD i INT) przesuwanie odbywa się na obydwóch bajtach. Dla typu INT znak liczby może ulec zmianie. Format tych instrukcji wygląda następująco:

<operand> <operator> <liczba przesunięć>

gdzie: <operand> stała numeryczna lub zmienna  
 <operator> LSH lub RSH  
 <liczba przesunięć> stała numeryczna lub zmienna określająca liczbę przesunięć

Przykłady poniżej ilustrują działanie operatorów LSH i RSH

(5)	00000101	(39)	00100111
(5 LSH 1 = 10)	00001010	(39 LSH 1 = 78)	01001110
(5 RSH 1 = 2)	00000010	(39 RSH 1 = 19)	00010011

działanie	MSB	LSB	\$
- - - -	01010110	11001010	\$56CA
LSH1	10101101	10010100	\$56CA LSH1 = \$AD94
RSH1	00101011	01100101	\$56CA RSH1 = \$2B65
LSH2	01011011	00101000	\$56CA LSH2 = \$5B28
RSH2	00010101	10110010	\$56CA RSH2 = \$15B2

Należy zwrócić uwagę, że LSH1 jest tym samym co pomnożenie przez 2, a RSH1 dzieleniem przez dwa. (dla liczb dodatnich).

Ten sposób mnożenia i dzielenia przez 2 jest szybszy niż instrukcje „+2” i „/2” ponieważ jest bliższy temu w jaki sposób wykonuje te operacje komputer i nie jest konieczna translacja wyrażenia na format binarny.

Operator „@” daje w wyniku adres zmiennej. Nie można go używać dla stałych numerycznych. „@ctr” da adres w pamięci zmiennej) „ctr”. Operator ten jest bardzo użyteczny w operacjach na zmiennych wskaźnikowych.

### 4.1.3 Operatory logiczne

Operatory te są dopuszczalne tylko w wyrażeniach logicznych, a te z kolei tylko w instrukcjach IF, WHILE, UNTIL.

=	sprawdzenie równości, np. 4 = 7 (fałsz)
#	sprawdzenie nierówności (różności), np.4 # 7 (prawda)
<>	j.w.
>	większy niż, np. 9>2 (prawda)
>=	większe lub równe, np. 5>=5 (prawda)
<	mniejszy niż, np. 2<9 (prawda)
<=	mniejsze lub równe, np. 5<=5 (prawda)
AND	logiczne „and” (zob.§4.4)
OR	logiczne „or” (zob. §4.4)

Operatory „#” i „<>”, oznaczają to samo i można ich używać wymiennie.

**UWAGA:** Kompilator *ACTION!* porównuje dwie wartości przez odjęcie ich od siebie i przyrównanie do 0. Metoda ta działa poprawnie z jednym wyjątkiem. Jeżeli porównywana jest duża liczba dodatnia typu `INT` z dużą liczbą ujemną typu `INT` wynik będzie nieprawidłowy (ponieważ typ `INT` używa najwyższego bitu jako bit znaku).

### 4.1.4 Priorytety operatorów

Sposób wartościowania wyrażenia zawierającego operatory jest określony przez ich priorytet przez „realizację operatora” rozumiemy wartościowanie kolejnych jego argumentów i zastosowanie go do wyliczonych wartości. Operatory o najwyższym priorytecie są realizowane najpierw.

W tablicy poniżej operatory są umieszczone w kolejności od najwyższego priorytetu do najniższego. Operatory w tej samej linii mają równe priorytety i są realizowane od lewej strony wyrażenia do prawej.

Zauważmy, że argumentem operatora może być wyrażenie ujęte w nawiasy okrągłe, co pozwala na zmianę kolejności wykonywanych operacji.

( )	nawiasy
- @	zmiana znaku liczby, adres
* / MOD LSH RSH	mnożenie, dzielenie itd.
+ -	dodawanie, odejmowanie
= # <> > >= < <=	operatory logiczne
AND &	„and” logiczne i ”and” działające na bitach
OR %	„or” logiczne i ”or” działające na bitach
XOR !	„exclusive or” działające na bitach

#### Przykład:

$4 + 5 * 3$  będzie obliczane jako  $4 + (5 * 3)$  ponieważ operator mnożenia ma wyższy priorytet niż „+”

wyrażenie	wynik	kolejność obliczeń
$4 / 2 * 3$	6	/, *
$5 < 7$	prawda	<
$43 \text{ MOD } 7 * 2 + 19$	21	MOD, *, +
$- ( ( 4 + 2 ) / 3$	-2	+, /, -

## 4.2 Wyrażenia arytmetyczne

Wyrażenia są zbudowane ze stałych, zmiennych i operatorów w taki sposób, że ich wykonanie daje w wyniku określoną wartość.

<operand> <operator> <operand>

gdzie: <operand> jest stałą liczbową, zmienną liczbową, wywołaniem funkcji (§6.2.3) lub innym wyrażeniem arytmetycznym.

Pierwsze trzy możliwości są zrozumiałe. Ostatnią z nich tłumaczy przykład poniżej:

– przeanalizujemy następujące wyrażenie  $3 * (4 + (22 / 7) * 2)$  sposób budowy tego wyrażenia zgodny z formatem zamieszczonym na poprzedniej stronie tłumaczy tabela:

Lp.	wyrażenie	wynik	wyrażenie uproszczone
1.	$(22 / 7)$	3	$3 * (4 + 3 * 2)$
2.	$(22 / 7) * 2$	6	$3 * (4 + 6)$
3.	$4 + (22 / 7) * 2$	10	$3 * 10$
4.	$3 * (4 + (22 / 7) * 2)$	30	30

Zwróćcie uwagę, że wyrażenia od 2 do 4 zawierają inne wyrażenie jako jeden ze swoich operandów.

Poniżej zamieszczone jest kilka innych przykładów. Słowa napisane małymi literami są zmiennymi lub stałymi.

wyrażenie	kolejność obliczeń
„A*(dog+7)/3 564	+, *, / stała jest także wyrażeniem
var & 7 MOD 3	MOD, &
ptr+@xyz	@, +

W wyrażeniach arytmetycznych mogą występować różne typy danych. Typ wynikowy wyrażenia zbudowanego z dwóch różnych typów przedstawia tablica:

	BYTE	INT	CARD
BYTE	BYTE	INT	CARD
INT	INT	INT	CARD
CARD	CARD	CARD	CARD

**UWAGA:** Po użyciu znaku „-” (zmiana znaku liczby) wynik jest typem INT, a po użyciu operatora adresu wynik jest typem CARD.

**UWAGA:** Do mnożenia używaj operandów typu INT, ponieważ dla bardzo dużych wartości CARD (>32767) wynik nie będzie poprawny.

### 4.3 Proste wyrażenia logiczne

Wyrażenia logiczne są wykorzystywane tylko w instrukcjach warunkowych. Służą do sprawdzania czy dane instrukcje powinny być wykonywane czy też nie. W prostych wyrażeniach logicznych może występować tylko jeden operator logiczny. Tworzenie bardziej skomplikowanych warunków jest omówione w §4.4.

Format prostych wyrażen logicznych jest następujący:

<wyrażenie arytmetyczne> <operator logiczny> <wyrażenie arytmetyczne>

**Przykład:**

```
@ cad <= $22A7
var <> y
5932 # counter
(5 & 7) * 8 >= (3 * (cad + dog))
addr / $FF + (@ptr + offset) <> $F03D - ptr & offset
(5+4) * 9 > ctr-1
```

### 4.4 Złożone wyrażenia logiczne

Złożone wyrażenia logiczne powstają przez połączenie prostych, wyrażen logicznych operatorami „AND” lub „OR”. Ich format ma następującą postać:

<wyrażenie log.> <operator spec.> <wyrażenie log.> |: <operator spec.> <wyrażenie log.> :|

gdzie: <wyrażenie log.> proste wyrażenie logiczne  
<operator spec.> jeden z operatorów „AND” lub „OR”

Tablica poniżej podaje wyniki działania operatorów „AND” i „OR”. „wyrażenie1” i „wyrażenie2” są prostymi wyrażeniami logicznymi.

		WYNIKI	
wyrażenie1	wyrażenie2	AND	OR
prawda	prawda	prawda	prawda
prawda	fałsz	fałsz	prawda
fałsz	prawda	fałsz	prawda
fałsz	fałsz	fałsz	fałsz

**UWAGA:** Część złożonego wyrażenia logicznego można ująć w nawiasy dla zapewnienia prawidłowej kolejności obliczeń. Jeżeli nie ma nawiasów, wyrażenie jest obliczane od strony lewej do prawej (zob. przykłady).

**UWAGA:** Jak już napisano w tym podręczniku, na poziomie bitów, operatory AND i & oraz OR i % są synonimami i są obliczane w ten sam sposób.

**Przykład:**

```

cat <= AND dog <> 13
(@ptr+7) * 3 # $60FF AND @ptr <= $1FFF
x ! $F0 <> 0 OR dog >= 100
(8 & cat) < 10 OR @ptr <> $0D
cat < 0 AND (dog > 400 OR dog <= 400)
ptr = $D456 OR ptr = $E000 OR ptr = $600

```

„Wyrażenie \$F0 AND \$0F jest fałszem ponieważ „AND” jest traktowane tutaj jako operator działający na bitach użyty w wyrażeniu arytmetycznym.

Wyrażenie \$F0 <> 0 AND \$0F00 jest prawdą ponieważ „AND” łączy w tym przypadku dwa proste wyrażenia logiczne i w ten sposób pełni rolę operatora złożonego wyrażenia logicznego.



## Rozdział 5 Instrukcje

Program komputerowy byłby bezużyteczny, jeśli by nie mógł aktywnie działać na danych. Do tej pory poznaliście jak deklorować zmienne, stałe itd., ale nie było mowy jak nimi manipulować. Instrukcje są aktywną częścią każdego języka programowania i *ACTION!* nie jest wyjątkiem. Instrukcje służą do zapisania określonego problemu formie, którą komputer rozumie i może poprawnie wykonać.

W *ACTION!* istnieją dwa rodzaje instrukcji: instrukcje proste i instrukcje strukturalne. Instrukcje prosta nie zawierają jako składowych innych instrukcji, natomiast instrukcje strukturalne zbudowane są na podstawie schematu strukturalizacji z ciągu instrukcji.

Instrukcje strukturalne można podzielić na dwie kategorie:

1. instrukcje warunkowe
2. instrukcje iteracyjne

### 5.1 Instrukcje proste

Instrukcje proste są tymi, które wykonują tylko pojedynczą czynność.

Są one podstawowymi elementami programu, ponieważ każde działanie, które wykonuje komputer jest instrukcją prostą. W *ACTION!* istnieją dwie instrukcje proste:

1. instrukcja przypisania (łącznie z wywołaniem funkcji)
2. instrukcja wywołania procedury

Instrukcje wywołania procedury i funkcji są omówione w rozdziale 6. Instrukcjami prostymi są również dwa słowa kluczowe:

1. EXIT omówiona w §5.2.3.2
2. RETURN omówiona w §6.1.2 i 6.2.2

#### 5.1.1 Instrukcja przypisania

Instrukcja przypisania służy do nadania zmiennej określonej wartości, forma tej instrukcji jest następująca:

`<zmienna> = <wyrażenie arytmetyczne>`

**UWAGA:** zmienna może być zmienną typu podstawowego lub tablicą, zmienną wskaźnikową lub polem rekordu. Wyrażenie musi być wyrażeniem arytmetycznym. Jeżeli zostanie użyte wyrażenie logiczne, zostanie wykryty błąd, ponieważ kompilator *ACTION!* nie przypisuje zmiennym liczbowym wyników wyrażeń logicznych.

Operatorem przypisania jest znak „=”. Wskazuje on komputerowi chęć nadania nowej wartości danej zmiennej. Znak „=”, jest również operatorem logicznym. Kompilator rozpoznaje jego prawidłowe znaczenie na podstawie kontekstu, w którym wystąpił.

Przykłady poniżej ilustrują sposób użycia instrukcji przypisania.

Należy zauważyć, że sekcja deklarująca zmienne poprzedza ich użycie, ponieważ niektóre przykłady pokazują co się stanie, gdy zmienna przypisywana jest innego typu.

BYTE b1, b2, b3, b4

INT i1

CARD d

b3=D w bajcie zarezerwowanym dla zmiennej b3 zostanie umieszczony kod ATASCII odpowiadający literze „D”

b4=\$44 liczba heksadecymalna \$44 zostanie umieszczona w bajcie przeznaczonym dla zmiennej b4 (\$44 jest kodem ATASCII litery D, tak więc zmienne b3 i b4 będą sobie równe).

b1 =b4+16 dodanie 16 do wartości zmiennej b4 i umieszczenie wyniku w bajcie zarezerwowanym na b1

c1=23439-\$0708 umieszczenie wartości 21431 (\$53B7) w dwóch bajtach zarezerwowanych dla zmiennej c1

i1=c1\*(-1) umieszczenie wartości 21431 (\$AC49) w dwóch bajtach zarezerwowanych na i1

b2=i1 umieszczenie wartości \$49 (73) w bajcie zarezerwowanym na b2. Zwróćcie uwagę, że komputer pobrał bajt LSB zmiennej i1 i umieścił pod b2 (MSB dla i1 wynosi \$AC, a LSB wynosi \$49).

b2=b2+1

zwiększenie wartości b2 o 1 i zapamiętanie z powrotem pod b2. Nowa wartość b2=\$4A (74).

Format ostatniego przykładu jest następujący:

<zmienna> = <zmienna> <operator> <operand>

ponieważ taki układ jest często stosowany, *ACTION!* dopuszcza skrócony zapis: :

<zmienna> == <operator><operand>

Operator musi być albo operatorem arytmetycznym albo działającym na poziomie bitów. Operand musi być wyrażeniem arytmetycznym. Przykłady poniżej ilustrują zastosowanie tego zapisu:

b2 == +1	oznacza	b2 = b2 + 1
b2 == -b1	oznacza	b2 = b2 - b1
b2 == & \$0F	oznacza	b2 = b2 & \$0F
b2 == LSH (5+3)	oznacza	b2 = b2 LSH (5+3)

Skrócona wersja tej instrukcji pozwala zaoszczędzić czas podczas wprowadzania programu, a ponadto generuje lepszy kod maszynowy w większości przypadków.

## 5.2 Instrukcje strukturalne

Gdyby istniały tylko instrukcje proste, ilość rzeczy jakie można by wykonać na komputerze byłaby bardzo ograniczona.

- aby powtórzyć grupę instrukcji pewną ilość razy należałoby wpisać je w tej samej kolejności żadaną ilość razy (dla 10 powtórzeń 10 instrukcji należałoby w programie umieścić aż 100 instrukcji).
- nie byłoby możliwe wykonywanie grupy instrukcji warunkowo tzn. tylko w określonych okolicznościach

Instrukcje strukturalne rozwiązują te i inne problemy. Można je podzielić na dwie kategorie: instrukcje warunkowe i instrukcje iteracyjne.

### 5.2.1 Instrukcje warunkowe

Instrukcja warunkowa pozwala testować określone wyrażenie logiczne i wykonywać różne instrukcje, zależnie od wyniku tego tekstu.

Istnieją trzy instrukcje warunkowe:

IF      WHILE      UNTIL

Instrukcje WHILE i UNTIL są instrukcjami iteracyjnymi i zostaną omówione później.

#### 5.2.1.1 Wyrażenia warunkowe

Wyrażenia te mogą przyjmować tylko dwie wartości: prawdę lub fałsz. Wyrażenia warunkowe nie są wyrażeniami nowego typu lecz są to albo wyrażenia logiczne, albo arytmetyczne. Tablica poniżej pokazuje wynik warunku dla różnych typów wyrażeń.

typ wyrażenia	wynik	interpretacja
arytmetyczny	zero (0)	fałsz
	0	prawda
logiczne	fałsz	fałsz
	prawda	prawda

#### 5.2.1.2 Instrukcja IF

Instrukcja IF (jeśli) ma podobne znaczenie jak w języku potocznym np. „jeżeli posiadam 9 dolarów lub więcej, kupię kawałek mięsa”.

W *ACTION!* to samo może wyglądać następująco:

```
BYTE  gotówka,
      mięso=[9],
      ryba=[8],
```

```

    kurczak=[6],
    hot-dog=[2]
IF gotówka >= 9 THEN
    kupić (mięso, gotówka)
FI

```

**UWAGA:** kupić (mięso, gotówka) jest wywołaniem procedury i zostanie o omówione w §6.1.3.”

Z powyższego przykładu można wywnioskować, że podstawową formą instrukcji IF jest:

```

IF <wyrażenie warunkowe> THEN instrukcja (-cje)>
FI

```

„FI” jest odwróconym słowem „IF” i oznacza koniec instrukcji IF.

Jest to konieczne, ponieważ po instrukcji IF może znajdować się cała lista instrukcji i kompilator musi wiedzieć które z nich mają być wykonywane jeżeli warunek jest spełniony.

Nie jest to jedyna forma instrukcji IF. Mogą wystąpić dwie opcje ELSE i ELSEIF. Weźmy następujące zdanie: „jeżeli posiadam 9 dolarów lub więcej, kupię sobie kawałek mięsa, w przeciwnym wypadku kupię rybę”

W *ACTION!* zostanie to zapisane jako:

```

IF gotówka >= 9 THEN
    kupić (mięso, gotówka)
ELSE
    kupić (ryba, gotówka)
FI

```

Kolejnym przykładem może być zdanie:

„jeżeli posiadam 9 dolarów lub więcej, kupię sobie mięso.

Jeżeli mam 8 do 9 dolarów kupię rybę.

Jeżeli mam 6 do 8 dolarów kupię kurczaka.

W przeciwnym wypadku kupię hot-doga”.

Można w tym przypadku zastosować instrukcję ELSEIF np.:

```

IF gotówka >= 9 THEN
    kupić (mięso, gotówka)
ELSEIF gotówka >=8 THEN
    kupić (ryba, gotówka)
ELSEIF gotówka >= 6 THEN
    kupić (kurczak, gotówka)
ELSE
    kupić (hot-dog, gotówka)
FI

```

Zwróćcie uwagę, że nie jest konieczne sprawdzanie, czy „gotówka >= 8 AND gotówka < 9” ponieważ komputer sprawdza tę listę sekwencyjnie z góry na dół. Jeżeli któryś z warunków jest prawdziwy, instrukcja której on odpowiada jest wykonywana, a dalsza część instrukcji IF (łącznie z ELSEIF i ELSE) jest opuszczana. Tak więc jeśli komputer przejdzie do warunku „gotówka >= 8”,automatycznie wiadomo, że na pewno jest ona mniejsza niż 9, ponieważ warunek poprzedni „gotówka >= 9” musiał być fałszywy.

## 5.2.2 Instrukcja pusta

Instrukcja pusta nie powoduje wykonania żadnych czynności. Wprowadzono ją do języka głównie z powodów syntaktycznych. Są jednak sytuacje w których możliwość użycia instrukcji pustej istotnie ułatwia opracowanie programu np. w pętlach czasowych i instrukcjach IF ... ELSEIF.

Ponieważ nie były do tej pory omawiane jeszcze instrukcje iteracyjne o pętlach czasowych powiemy tylko tyle, że są wykorzystywane do „tracenia czasu” np. jeżeli chce się zrobić przerwy czasowe pomiędzy drukowaniem na ekranie poszczególnych linii należy zastosować pętlę czasową oczekującą na stosowny moment.

Przykład poniżej ilustruje instrukcję pustą dla IF ... ELSEIF.

**Przykład:** piszemy program, który ma pozwolić maklerowi giełdowemu odnaleźć informację o określonym kapitale przy użyciu pewnych komend. Makler może użyć następujących poleceń: BUY, DOWN?, FIND, QUIT, SELL i UP?, z tym, że nie wiadomo jeszcze na czym będzie polegało wykonanie komendy FIND. Program testuje pierwszą literę polecenia i podejmuje określone działanie. Mimo, że polecenie FIND nie jest określone można rozpocząć testowanie programu. W części dotyczącej wykonania polecenia FIND należy umieścić instrukcję pustą, która w przyszłości zostanie zastąpiona właściwymi. Odpowiedni fragment programu przyjmie następującą postać:

```
IF chr=„B THEN
  dobuy()
ELSEIF chr=„D THEN
  dodown()
ELSEIF chr=„F THEN
  ; ++++ tutaj znajduje się instrukcja pusta
ELSEIF chr=„Q THEN
  doquit()
ELSEIF chr=„S THEN
  dosell()
ELSEIF chr=„U THEN
  doup()
ELSE
  doerror( ) ; ++++ polecenie nierozpoznane
FI
```

Wszystkie instrukcje „do...” są wywołaniami procedur, które wykonują określone działania dla poszczególnych poleceń.

Dla polecenia FIND nie jest podejmowane żadne działanie. Jeżeli polecenie to zostanie w przyszłości określone, wystarczy w miejsce instrukcji pustej umieścić „dofind()”, a w innym miejscu programu rozpisać całą procedurę „dofind”.

## 5.2.3 Pętle

Pętle są używane do wielokrotnego wykonywania pewnej grupy instrukcji. Np. jeżeli z pewnych powodów ekran ma zostać wypełniony gwiazdkami, można albo użyć w tym celu szeregu pojedynczych instrukcji wyświetlających na ekranie po jednej gwiazdce, albo zastosować do tego instrukcji iteracyjnej. Wystarczy wtedy określić tylko ile razy pojedyncza gwiazdka ma być drukowana na ekranie i zastosować odpowiednią instrukcję.

Liczbę powtórzeń można określić na dwa sposoby. Pierwszy z nich to: podanie wprost odpowiedniej liczby, drugi to umieszczenie wyrażenia warunkowego określającego liczbę wykonań pętli.

Instrukcją FOR używa pierwszej metody, natomiast WHILE i UNTIL drugiej.

Jeżeli nie: byłaby określona liczba powtórzeń lub umieszczony warunek nigdy nie osiągnąłby wartości oznaczającej koniec obliczeń w danej pętli, pętla wykonywana byłaby nieskończoną ilość razy. Jedynym sposobem wyjścia z takiej pętli jest naciśnięcie klawisza <SYSTEM RESET>

Pętle w *ACTION!* zbudowane są na następującej zasadzie:

- pętla podstawowa, która jeżeli jest użyta osobno jest pętlą nieskończoną,
- instrukcje sterujące pętlą (FOR, WHILE, UNTIL), które, ograniczają liczbę powtórzeń pętli podstawowej.

### 5.2.3.1 DO i OD

„DO” i ”OD” są używane, do oznaczenia początku i końca pętli podstawowej. Wszystkie instrukcje znajdujące się pomiędzy nimi są częścią danej pętli. Jak już wspomniano pętla podstawowa bez instrukcji sterujących jest pętlą nieskończoną. Program poniżej ilustruje użycie pętli „DO-OD”

**Przykład 1:**

```
PROC timestwo()
  CARD i=[0],j
```

```

DO                                ; początek pętli DO-OD
  i==+1                          ; zwiększenie i o 1
  j=i*2                          ; j równa się i*2
  PrintC (i)
  Print („razy 2 równa się”)
  PrintCE (j)
OD                                ; koniec pętli DO-OD
RETURN

```

**UWAGA:** Instrukcje. `PrintC`, `Print`, `PrintCE` są procedurami bibliotecznymi. Dokładny ich opis znajduje się w części VI podręcznika. Procedury te powodują wyprowadzenie wartości odpowiedniej zmiennej lub napisu na ekran.

Wynik działania programu 1:

```

1 razy 2 równa się 2
2 razy 2 równa się 4
3 razy 2 równa się 6
4 razy 2 równa się 8
5 razy 2 równa się 10
6 razy 2 równa się 12
7 razy 2 równa się 14
8 razy 2 równa się 16
...

```

Kropki na końcu pokazują, że będzie to trwało wiecznie lub do momentu naciśnięcia klawisza <SYSTEM RESET>. Sama pętla DO-OD jest mniej więcej bezużyteczna, ale w połączeniu z instrukcjami sterującymi pętlą FOR, WHILE i UNTIL staje się jedną z najbardziej przydatnych dostępnych instrukcji.

**UWAGA:** naciśnięcie klawisza <BREAK> spowoduje również wyjście z pętli w przykładzie nr 1, ponieważ pętla wykonuje wiele operacji we/wy. (<BREAK> działa tylko przy wykonywaniu dużej liczby operacji we/wy. Więcej informacji można znaleźć w części IV („Kompilator ACTION!”).

Ileokroć zobaczysz pętlę „<DO-OD>”, w formatach instrukcji sterujących pętlą, pamiętaj, że oznacza to pętlę, a to z kolei oznacza parę DO-OD otaczającą pętlę.

### 5.2.3.2 Instrukcja EXIT

Instrukcja EXIT służy do płynnego wychodzenia z dowolnej pętli. Ta instrukcja spowoduje, że wykonanie programu przejdzie do instrukcji następującej po następnym „OD”. Oto przykład:

**Przykład 1:**

```

PROC timestwo()
  CARD i=[0],j
  DO                                ; początek pętli DO - OD
    i==+1                          ; zwiększenie i o 1
    j=i*2                          ; j równa się i*2
    PrintC(i)
    Print(„ razy 2 równa się „)
    EXIT                            ; instrukcja EXIT
    PrintCE(j)
  OD                                ; koniec pętli DO-OD
  ; kontynuowane programu po „EXIT”
  PrintEC „Koniec tablicy”)
RETURN

```

Wynik działania programu 1:

```

1 razy 2 równa się 2   Koniec tabeli

```

Jak widać, instrukcja „`PrintCE(j)`” nigdy nie jest wykonywana. Instrukcja EXIT wymusza przeskok do instrukcji „`PrintE(„Koniec tabeli”)`”. EXIT nie jest zbyt przydatny, gdy jest używany samodzielnie, ale jeśli użyjesz go w połączeniu z instrukcją IF (tj. zamienisz EXIT w warunkowy skok z pętli), może być bardzo przydatny, jak pokazuje program poniżej.

**Przykład 2:**

```

PROC timestwo()
  CARD i=[0],j
  DO                                     ; początek pętli DO-OD
  IF i=15 THEN                           ; EXIT po spełnieniu warunku w IF
    EXIT
  FI
  I==+1                                   ; zwiększenie i o 1
  J=i*2                                   ; j równa się i*2
  PrintC(i)
  Print(„razy 2 równa się „)
  PrintCE(j)
  OD                                     ; koniec pętli DO-OD
  ; kontynuowanie programu, gdy i=15
  PrintE(„Koniec Tabeli“)
RETURN

```

**Wynik działania programu:**

```

1 razy 2 równa się 2
2 razy 2 równa się 4
3 razy 2 równa się 6
4 razy 2 równa się 8
5 razy 2 równa się 10
6 razy 2 równa się 12
7 razy 2 równa się 14
8 razy 2 równa się 16
9 razy 2 równa się 18
10 razy 2 równa się 20
11 razy 2 równa się 22
12 razy 2 równa się 24
13 razy 2 równa się 26
14 razy 2 równa się 28
15 razy 2 równa się 30   Koniec tabeli

```

To użycie zamienia nieskończony blok pętli w skończony. EXIT może sterować wykonaniem pętli, ale nie jest uważany za instrukcję kontrolującą pętlę strukturalną, ponieważ nie jest samodzielny; oznacza to, że jest użyteczny tylko w połączeniu ze strukturalną instrukcją „IF”.

## 5.2.4 Sterowanie pętlą

*ACTION!* ma trzy instrukcje strukturalne, które kontrolują podstawową pętlę DO-OD:

1. FOR
2. WHILE
3. UNTIL

Mówiąc, że „kontrolują podstawową pętlę DO-OD”, mamy na myśli, że ograniczają liczbę wykonań nieskończonej pętli, czyniąc ją w ten sposób pętlą skończoną. Sterowanie pętlami to jedno z urządzeń czyniących komputery bardzo użytecznymi. Gdyby ktoś tysiąc razy kazał ci napisać „Nigdy więcej nie będę pluć”, nazwałbyś to karą, ale gdybyś kazał komputerowi zrobić to samo (oczywiście w kontrolowanej pętli), pomyślałby, że zadanie było łatwe i przyjemne.

Teraz przyjrzymy się szczegółowo każdej instrukcji sterującej pętlą, a następnie zajmiemy się właściwością wszystkich instrukcji strukturalnych *ACTION!*: zagnieżdżaniem.

### 5.2.4.1 Instrukcja FOR

Instrukcja FOR służy do powtórzenia pętli określoną liczbę razy. Wymaga własnej, specjalnej zmiennej, powszechnie zwanej licznikiem. W przykładach licznik będzie nosił nazwę „ctr”, aby ci o tym przypomnieć, ale możesz go nazwać jak chcesz. Format instrukcji FOR jest następujący:

```

FOR <licznik> = <wartość początkowa> TO <wartość końcowa>
                                     (KROK <przyrost>) <pętla DO-OD>

```

gdzie:

<licznik>	zmienna używaną do przechowywania liczby wykonań pętli
<wartość początkowa>	wartością początkową licznika
<wartość końcowa>	to wartość końcowa licznika
<przyrost>	liczba o jaką jest zwiększany licznik przy każdej iteracji
<pętla DO-OD>	pętla podstawowa DO-OD

**UWAGA:** Parametr „STEP (KROK) <przyrost>„ jest opcjonalny

Zamiast tłumaczyć Ci to metaforami, podrzucimy Ci kilka przykładów, bo one mniej więcej mówią same za siebie. Po każdym z nich znajduje się jego wynik.

### Przykład 1:

```
PROC hithere()
    BYTE ctr                                ; licznik używany w pętli FOR
    FOR ctr=1 TO 5                          ; pętla FOR bez parametru „STEP” – domyślnie „1”
        DO
            PrintE („Cześć”)
        OD
RETURN
```

Wynik działania programu 1:

```
Cześć
Cześć
Cześć
Cześć
Cześć
Cześć
```

### Przykład nr 2:

```
PROC evens()
    BYTE ctr                                ; licznik używany w pętli FOR
    FOR ctr=0 TO 16 STEP 2                  ; ta pętla FOR zawiera „STEP”
        DO
            PrintB(ctr)
            Print („ ")
        OD
RETURN
```

Wynik działania programu 2:

```
0 2 4 6 8 10 12 14 16
```

Przyjrzyj się formatowi instrukcji FOR. Zauważ, że nie powiedzieliśmy nic o używaniu zmiennych numerycznych jako <wartość początkowa>, <wartość końcowa> lub <przyrost>. Jest to dozwolone i pozwala na wykonanie pętli FOR zmienną liczbę razy.

Jeśli zmienisz wartość zmiennych używanych jako <wartość początkowa>, <wartość końcowa> lub <przyrost> w samej pętli, nie zmienisz liczby wykonań pętli. Jest to prawda, ponieważ <wartość początkowa>, <wartość końcowa> i <przyrost> mają stałą wartość po wejściu do pętli. Jeśli używasz zmiennych, wartością używaną podczas ich ustawiania jest wartość, jaką miała zmienna w momencie pierwszego wejścia do pętli.

### Przykład nr 3:

```
PROC changeloop()
    BYTE ctr,
    start=[1],
    end=[50]
    FOR ctr = start TO end
        DO
            start = 100                    ; nie powoduje zmiany ilości iteracji
            end =10                         ; nie powoduje zmiany ilości iteracji
            PrintBE (ctr)
```

```

ctr- ==i*2
OD
RETURN

```

; powoduje zmianę ilości iteracji

Wynik działania programu 3:

```

1
3
7
15
31

```

Tablica poniżej ilustruje co się dzieje podczas każdej iteracji. Kolumna ostatnia pokazuje; zmiany licznika po instrukcji przypisania.

iteracja	licznik	Print	ctr==*2
1	1	1	2
2	3	3	6
3	7	7	14
4	15	15	30
5	31	31	62

Po piątej iteracji wartość licznika wynosi 62. Jest to wartość większa od założonej wartości końcowej licznika (50) i dlatego ilość iteracji wynosi 5, a nie jakby się mogło wydawać 50.

Zmiana wartości licznika wewnątrz pętli prowadzi do bardzo interesujących wyników. Należy jednak zachować przy tym dużą ostrożność.

**Przykład pętli czasowej:**

```

BYTE ctr
FOR ctr=1 TO 250
DO
; instrukcja pusta *
OD

```

Jest to typowe użycie pętli FOR i instrukcji pustej do odczekania pewnego przedziału czasu. Jest to przydatne przy pisaniu gier.

**UWAGA:** Jeżeli <wart. końcowa> pętli będzie większa niż dopuszcza to użyty typ licznika (tzn. „FOR ctr=0 TO 255” jeżeli ctr jest typu BYTE lub „FOR ctr=0 TO 65535” gdy ctr jest typu CARD), pętla ta stanie się pętlą nieskończoną, ponieważ licznik nie może być zwiększany ponad wartość maksymalną dla danego typu.

#### 5.2.4.2 Instrukcja WHILE

Instrukcja WHILE (oraz UNTIL) jest używana w przypadkach, gdy nie jest z góry określona ilość iteracji. Wykonywanie instrukcji wewnątrz pętli powtarzane jest tak długo, jak długo spełniony jest odpowiedni warunek. Format tej instrukcji, ma następującą postać: „

```

WHILE <wyrażenie warunkowej
<pętla DO-OD>

```

Ponieważ sprawdzanie warunku odbywa się na początku, jeżeli wartość tego wyrażenia przed rozpoczęciem iteracji jest fałszem, to instrukcje wewnętrzne nie są wykonywane ani razu i realizacja pętli zostaje zakończona.

**Przykład nr 1:**

```

PROC factorials()
; ++++ procedura ta drukuje wartości silnii, aż do zadanej z góry granicy
CARD fact=[1], ; wartość silnii zmiennej num
num=[1], ; licznik
amt=[6000] ; górna granica obliczeń
Print („Silnie mniejsze niż „)
PrintC (amt) ; drukowani górnej granicy obliczeń
PrintE ([:]) i drukuje znak „:” i powoduje przejście
; do nowej linii
PutE() ; przejście do nowej linii

```



```

WHILE fact * num < amt      ; sprawdzenie wartości następnej silnii
  DO                        ; początek pętli
  fact==*num
  PrintC (num)              ; drukuje liczbę dla której jest
                            ; obliczana silnia
  Print („ silnia wynosi „)
  PrintCE (fact)           ; drukuje wartość silnii
  num==+1                  ; zwiększenie liczby o 1
  OD                        ; koniec pętli WHILE
RETURN

```

### Wynik działania programu 1:

```

Silnie mniejsze niż 6000:
1 silnia wynosi 1
2 silnia wynosi 2
3 silnia wynosi 6
4 silnia wynosi 24
5 silnia wynosi 120
6 silnia wynosi 728
7 silnia wynosi 5040

```

**UWAGA:** Podając zagranicę obliczeń silni liczbę 40000 okaże się, że kompilator nie sprawdza błędu nadmiaru. Po osiągnięciu liczby większej niż pozwala typ CARD (65535) następuje start od 0. Dla liczby 65000 na wyjściu uzyska się 66000–65536=464 ponieważ zmienna była zwiększana do dopuszczalnej granicy, następnie została wyzerowana i ponownie zwiększana. Więcej informacji na ten temat znajduje się w części IV podręcznika poświęconej kompilatorowi *ACTION!*

### Przykład nr 2:

```

PROC guesswhile()
;++++ zabawa w odgadywanie liczby napisana za pomocą pętli WHILE
  BYTE num,                      ; liczba do odgadnięcia
      guess=£2003                 ; zmienna guess jest ustawiona na wartość
                                  ; z poza zakresu określonego regułami gry

  PrintE („Zapraszam do wzięcia udziału w grze.“)
  PrintE („Spróbuj odgadnąć liczbę z przedziału 0-100.“)
  num = Rand (101)                ; generuje liczbę do odgadnięcia
  WHILE guess<>num                ; początek pętli WHILE
  DO                               ;
  Print („Jaki jest Twój typ? „)
  guess = InputB()                ; wprowadzenie odpowiedzi użytkownika
  IF guess < num THEN              ; liczba za mała
    PrintE („Zbyt niski, próbuj ponownie.“)
  ELSEIF guess > num THEN          ; liczba za duża
    PrintE („Zbyt wysoki, próbuj ponownie.“)
  ELSE
    PrintE („Gratulacje!!!!!!“)
    PrintE („Odgadłeś.“)
  FI                               ; koniec sprawdzania odpowiedzi
  OD                               ; koniec pętli WHILE
RETURN ; koniec procedury

```

### Wynik działania programu 2:

```

Zapraszam do wzięcia udziału w grze.
Spróbuj odgadnąć liczbę z przedziału 0-100.
Jaki jest twój typ? 50
Zbyt niski, próbuj ponownie.
Jaki jest twój typ? 60
Zbyt wysoki, próbuj ponownie.
Jaki jest twój typ? 55
Zbyt niski, próbuj ponownie.
Jaki jest twój typ? 57
Gratulacje!!!!!!
Odgadłeś.

```

Zwróć uwagę jak mogą być użyteczne instrukcje IF wewnątrz pętli.

### 5.2.4.3 Instrukcja UNTIL

W paragrafie poprzednim było powiedziane, że pętla WHILE może nie : zostać wykonana ani razu, ponieważ wyrażenie warunkowe jest obliczalne przed wejściem w pętlę .W instrukcji UNTIL warunek umieszczony jest po każdym obiegu pętli, tak więc musi być ona przynajmniej raz wykonana. Format:

```
DO
<instrukcja>
.
.
.
<instrukcja>
UNTIL <wyrażenie warunkowe>
OD
```

Jeżeli wyrażenie warunkowe jest prawdą, wówczas wykonywanie . będzie kontynuowane począwszy od pierwszej instrukcji po „OD”.

#### Przykład nr 1:

```
PROC guessuntil()
; ++++ zabawa w odgadywanie liczby napisana za pomocą pętli UNTIL
  BYTE num,
    guess
  PrintE („Zapraszam do wzięcia udziału w grze.”)
  PrintE („Spróbuj odgadnąć liczbę z przedziału 0-100.”)
  num = Rand(101)
  DO
  Print („Jaki jest twój typ? „)
  guess = InputB()
  IF guess < num THEN
    PrintE ( „Zbyt niski, próbuj ponownie.”)
  ELSEIF guess > num THEN
    PrintE („Zbyt wysoki, próbuj ponownie.”)
  ELSE
    PrintE („Gratulacje!!!!”)
    PrintE („Odgadłeś.”)
  FI
  UNTIL guess = num
  OD
RETURN
```

Przykład ten jest identyczny z poprzednim, ale został napisany z pomocą pętli UNTIL. Zwróćcie uwagę, że zmienna „guess” nie była inicjowana w deklaracji typu, tak jak to było dla pętli WHILE. Jest to możliwe, ponieważ wyrażenie „guess = num” jest obliczane dopiero po wprowadzeniu pierwszej odpowiedzi przez użytkownika.

## 5.2.5 Zagnieżdżanie instrukcji strukturalnych

Instrukcja strukturalna łączy jedną lub więcej instrukcji według pewnego schematu strukturalizacji. Schemat ten określa, czy instrukcje wewnętrzne mają być wykonywane kolejno, czy mają być wykonane tylko niektóre (lub żadna) z nich zależnie od wartości określonych warunków, czy też mają być wykonywane wielokrotnie. Instrukcje wewnątrz instrukcji strukturalnej mogą być albo instrukcjami prostymi, albo innymi instrukcjami strukturalnymi. Umieszczenie jednej instrukcji strukturalnej wewnątrz innej nazywa się „zagnieżdżaniem”.

W §5.2.4.2 i §5.2.4.3 w przykładach można było zobaczyć zagnieżdżanie instrukcji IF w pętlach WHILE i UNTIL.W tym paragrafie omówimy bardziej skomplikowane konstrukcje.

Jeżeli instrukcje IF są umieszczone w innych instrukcjach IF, może się wydawać, że trudno będzie określić, które ELSE należy do którego IF. Kompilator rozstrzyga ten problem dzięki parom IF-FI. FI stanowi parę dla najbliższego poprzedzającego go IF, który nie był do tej pory połączony z innym FI.

```
IF <wyrażenie A> THEN
↑
  IF <wyrażenie B> THEN
  ↑↓
  <instrukcje
  ↓
  ELSEIF <wyrażenie C> THEN           ; ELSEIF do IF <wyrażenie B>
  ↑
  IF<wyrażenie D> THEN
  ↑↓
```

```

        <instrukcje>
        ELSE                               ; ELSE do IF <wyrażenie D>
        ↓ <instrukcje>
        FI                                 ; koniec IF <wyrażenie D>
    FI                                     ; koniec IF <wyrażenie B>
ELSEIF <wyrażenie E> THEN                ; ELSEIF do IF <Wyrażenie A>
    ↓ <instrukcje>
ELSE                                       ; ELSE do IF <wyrażenie A>
    ↓ <instrukcje>
FI                                         ; koniec IF <wyrażenie A>

```

Linie przerywane łączą pary IF-FI.

Program następnym pokazuje zagnieżdżanie pętli FOR i ma na celu wydrukowanie tabliczki mnożenia 10\*10.

```

PROC timestable()
;++++ procedura ta drukuje na ekranie tabliczkę mnożenia 10*10
    BYTE c1,                               ; licznik zewnętrznej pętli FOR
        c2                                 ; licznik wewnętrznej pętli FOR
    FOR c1 = 1 TO 10
        DO                                  ; początek pętli zewnętrznej
        IF c1<10 THEN                       ; dla cyfr od 1-9 trzeba pozostawić
            Print ( " ")                   ; jedno wolne miejsce
        FI
        PrintB(c1)                          ; drukowanie pierwszej liczby w kolumnie
        FOR c2 = 2 TO 10
            DO                                ; początek pętli wewnętrznej
            IF c1*c2<10 THEN                 ; dla liczb jednocyfrowych należy
                Print ( " ")               ; zostawić trzy wolne miejsca
            ELSEIF c1*c2<100 THEN           ; dla liczb dwucyfrowych należy
                Print ( " ")               ; zostawić dwa wolne miejsca
            ELSE                             ; dla liczb trzycyfrowych należy
                Print ( " ")               ; zostawić tylko jedno wolne miejsce
            FI
            PrintB (c1*c2)                  ; drukowanie wyniku
        OD                                  ; koniec pętli wewnętrznej
        PutE                                ; przejście do następnej linii
    OD                                       ; koniec pętli zewnętrznej
RETURN                                       ; koniec procedury

```

Wynik działania programu:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

## Rozdział 6 Procedury i funkcje

Rozwiązując bardziej złożony problem wyodrębnia się zwykle, pewna jego części i oddzielnie formułuje się dla nich rozwiązania. Można przy tym najpierw rozwiązywać podproblemy, a następnie łączyć je w całość, albo też odwrotnie tzn. formułować rozwiązanie całego zagadnienia w terminach nie rozpatrzonych jeszcze dotąd części, a dopiero później, schodzić na niższy poziom szczegółowości. W programowaniu strukturalnym stosuje się zwykle to drugie podejście. Wydzielanie podproblemów ma istotne zalety, gdyż umożliwia prowadzenie rozumowania na ustalonych poziomach abstrakcji.

We wszystkich językach programowania istnieją mechanizmy ułatwiające dzielenie rozwiązywanego problemu na części. Mechanizmy takie w języku *ACTION!* to przede wszystkim procedury i funkcje.

Prawie każdą czynność, którą się wykonuje można nazwać procedurą lub funkcją np.:

Procedury:	Funkcje:
mycie samochodu przygotowywanie obiadu pójście do szkoły	wyprowadzania salda swojej książeczki czekowej szukanie numeru telefonu

„mycie samochodu” w rzeczywistości składa się z szeregu drobnych czynności, takich jak podłączenie węża do kranu, przygotowanie środków myjących itp.

W języku programowania jest podobnie. Można łączyć grupę prostych działań, które wspólnie rozwiązują jedno większe zadanie w procedurę lub w funkcję i nadać jej nazwę. Jeżeli następnie chce się wykonać dane zadanie, wystarczy użyć nazwy tej procedury lub funkcji. Jest to tak zwane wywołanie procedury lub funkcji. Zanim dana procedura lub funkcja zostanie wywołana, musi być wcześniej zdefiniowana.

Jaka jest różnica pomiędzy procedurami i funkcjami? Tak jedne, jak i drugie wykonują serię kroków aby rozwiązać dane zadanie. W wyniku działania procedury można uzyskać szereg (nie tylko jedną) wartości. Funkcja wyznacza natomiast tylko pojedynczą wartość. Dlatego też wywołanie funkcji odbywa się nie za pomocą osobnej instrukcji, lecz bezpośrednio w wyrażeniach, w których chcemy skorzystać z wartości tej funkcji.

W przykładach pochodzących z życia użyto jako funkcji „szukanie numeru telefonu” ponieważ czynność ta ma na celu odnalezienie, dokładnie jednej wartości, która następnie będzie użyta przy dzwonieniu.

**UWAGA:** W dalszej części tego podręcznika będziemy, używać określenia „podprogram” zamiast „procedura” lub „funkcja”. Jeżeli natomiast będzie użyte słowo „procedura” lub „funkcja”, oznaczać to będzie, że omawiana cecha dotyczy wyłącznie określonej klasy podprogramów, a dla drugiej nie ma zastosowania.

### 6.1 Procedury

Procedury są używane aby zgrupować instrukcje, które wykonują określone zadanie, w jeden blok mający swoją własną nazwę i mogący być następnie wywołany do wykonania tego zadania. Aby zastosować procedury trzeba wiedzieć jak:

1. zadeklarować procedurę
2. wywołać procedurę

Kolejne trzy paragrafy wyjaśniają jak to zrobić oraz podają przykłady procedur.

#### 6.1.1 Deklaracja procedury

Słowo kluczowe „PROC” jest używane do zapisania początku deklaracji procedury. Cała procedura składa się z grupy instrukcji, pewnych informacji początkowych oraz instrukcji RETURN umieszczonej na końcu.

```
PROC <identyfikator> {=<adres>^} ({<lista parametrów>})
  {<deklaracja zmiennych>}
  {<lista instrukcji>}
RETURN
```

gdzie:

PROC

jest słowem kluczowym języka *ACTION!* i oznacza początek deklaracji procedury

<identyfikator>	jest nazwą procedury
<adres>	opcjonalnie określa adres początkowy procedury (zob. 9.3)
<lista parametrów>	lista parametrów żądanych przez procedurę
<deklaracja zmiennych>	lista zmiennych lokalnych dla procedury (zob. 3.4.1 oraz 6.3)
<lista instrukcji>	instrukcje znajdujące się w procedurze
RETURN	oznacza koniec procedury

**UWAGA:** <lista parametrów>, <deklaracja zmiennych> oraz <lista instrukcji> są opcjonalne. Prawdopodobnie przynajmniej jedno z nich będzie użyte, ale poniższa deklaracja procedury jest również poprawna:

```
PROC nothing ( ) ; nawiasy są obowiązkowe
RETURN
```

Procedura ta nie powoduje wykonania żadnej czynności, ale ten typ procedur pustych jest bardzo użyteczny podczas pisania programów.

Jeżeli mamy napisany program, który wywołuje procedurę o nazwie „dotest”, ale procedura ta nie jest jeszcze gotowa, można użyć procedury pustej. Dzięki temu można testować program bez niebezpieczeństwa, że zostanie wykryty błąd „undeclared variable” (niezdefiniowana zmienna).

Pojęcia <lista parametrów> i ”RETURN” zostaną wytłumaczone później.

Wróćmy do przykładu z §5.2.4.3. Obecnie wiadomo już skąd się wzięła w tym programie instrukcja PROC i deklaracja zmiennych. W *ACTION!* deklaracje procedur i funkcji mogą być kompilowane oddzielnie.

Przykład ten jest deklaracją procedury, a więc również poprawnym programem i jako taki może być skompilowany i uruchomiony.

## 6.1.2 RETURN

RETURN ma na celu wskazać kompilatorowi, że ma opuścić procedurę i przekazać sterowanie w miejsce gdzie nastąpiło jej wywołanie. Wykonywanie programu będzie kontynuowane od linii następnej. Jeżeli skompilowana była pojedyncza procedura (program złożony z jednej procedury) sterowanie będzie przekazane do monitora *ACTION!*

**UWAGA:** Kompilator nie wykrywa braku instrukcji RETURN.

W procedurze może występować więcej niż jedna instrukcja RETURN. Np. jeżeli w procedurze znajduje się instrukcja IF z wieloma ELSEIF można użyć RETURN po jednym lub więcej ELSEIF. Ilustruje to następujący przykład:

```
PROC testcomraand ( )
;++++ procedura ta testuje poprawność wprowadzonej komendy.
; Komendy poprawne to 0, 1, 2, 3. Jeżeli komenda jest inna
; drukowany jest komunikat o błędzie i sterowanie powraca
; w miejsce skąd była wywołana ta procedura.

BYTE cmd
Print („Komenda? „)
cmd = InputB()
IF cmd > 3 THEN
    PrintE („Komenda błędna!!“)
    RETURN ; wyjście z podprogramu
ELSEIF cmd = 0 THEN
    <instrukcja 0>
ELSEIF cmd = 1 THEN
    <instrukcja 1>
ELSEIF cmd = 2 THEN
    <instrukcja 2>
ELSEIF cmd = 3 THEN
    <instrukcja 3>
FI
RETURN
```

## 6.1.3 Wywołanie procedury

W programach poznanych do tej pory były już zastosowane instrukcje wywołania procedury chociaż możecie nawet o tym nie wiedzieć.

W każdym przypadku, gdy korzystaliśmy z podprogramów bibliotecznych następowało wywołanie procedury. Format:

```
<identyfikator> ({<lista parametrów>})
```

gdzie:           <identyfikator>                   nazwa procedury, którą się wywołuje  
                  <lista parametrów>               zawiera wartości, które chce się przekazać do procedury

### Przykłady:

```
PrintE („zapraszam do wzięcia udziału w grze.”)
PrintE („Spróbuj odgadnąć liczbę z przedziału 0-100.”)

factorials()
guessuntil()

BYTE z
CARD add
signoff (add, z)
```

Zanim zostaną wywołane procedury „factorials”, „guessuntil”, „signoff” wcześniej muszą być zadeklarowane. „PrintE” jest podprogramem bibliotecznym, tak więc nie trzeba go deklarować ponieważ jest już zadeklarowany w bibliotece *ACTION!*

Należy pamiętać, że nawiasy są konieczne nawet gdy procedura nie posiada parametrów. Ilość parametrów przy wywołaniu procedury nie może być większa (ale może być mniejsza) niż ilość parametrów określonych w deklaracji procedury (zob. §6.4.).

## 6.2 Funkcje

Jak już zauważono, podstawową różnicą pomiędzy procedurami i funkcjami jest to, że w wyniku działania funkcji uzyskuje się pojedynczą wartość. Deklaracja i wywołanie funkcji ma nieznacznie inną formę niż dla procedur. Ponieważ funkcje, dają wartości liczbowe, muszą być stosowane tam gdzie użycie liczby jest poprawne z punktu widzenia języka (np. w wyrażeniach arytmetycznych).

### 6.2.1 Deklaracja funkcji

Deklaracja funkcji jest bardzo podobna do deklaracji procedury. Jediną różnicą jest to, że należy określić jakiego typu jest wartość obliczana przez funkcję (BYTE, CARD lub INT) oraz co to jest za wartość.

```
<typ> FUNC <identyfikator> {=<adres>} ({<lista parametrów>})
          {<deklaracja zmiennych>}
          {<lista instrukcji>}
RETURN (<wyrażenie arytmetyczne >)
```

gdzie:           <typ>                                   typ danych (podstawowy) dla wartości otrzymywanej z funkcji  
                  FUNC                                   słowo kluczowe” języka oznaczające deklarację funkcji  
                  <identyfikator>                   nazwa funkcji  
                  <adres>                           opcjonalnie określa początkowy adres funkcji (zob. 9.3)  
                  <lista parametrów>               lista parametrów żądanych przez; funkcję (zob. 6.4)  
                  <deklaracja zmiennych>       lista zmiennych lokalnych danej funkcji (zob. 3.4.1 i 6.5)  
                  <lista instrukcji>               ciąg instrukcji stanowiących funkcję  
                  RETURN                           oznacza koniec funkcji  
                  <wyrażenie arytmetyczne>      wartość, która jest otrzymywana w wyniku działania funkcji

Podobnie jak dla deklaracji procedury, <lista parametrów>, <deklaracja zmiennych> oraz <lista instrukcji> są opcjonalne. W przypadku procedur, opuszczając je otrzymuje się procedurę pustą. Dla funkcji wygląda to trochę inaczej:

**Przykład 1:**

```
CARD FUNC square (CARD x)
RETURN (x*x)
```

Funkcja ta wyznacza kwadrat wprowadzonej zmiennej x. Wartość, która jest otrzymywana z funkcji, jest zapisywana w formie wyrażenia. W przykładzie 1 jest to „x\*x”.

W **przykładzie 2** wyrażenie arytmetyczne jest po prostu nazwą zmiennej.

```
BYTE FUNC getcommand()
; ++++funkcja ta odczytuje nr polecenia i jeżeli jest on poprawny
; kończy działanie. Jeżeli podany numer jest błędny
; (tzn. mniejszy niż 1 lub większy od 7),
; wyprowadzany jest komunikat o błędzie.
BYTE command,                ; zmienna, w której znajduje się wprowadzany nr polecenia
    error                    ; ustawiany na 1, gdy komenda jest błędna
DO
Print ("POLECENIE? „)
command = InputB()
IF command < 1 OR command > 7 THEN          ; polecenie błędne
    error = 1
    PrintE („Polecenie błędne, spoza zakresu 1-7“)
ELSE                                         ; polecenie poprawne
    error = 0
FI
OD
RETURN (command)
```

**UWAGA:** wyrażenie arytmetyczne w instrukcji RETURN musi się znajdować nawiasach.

Powyższe przykłady są stosunkowo proste. Funkcje mogą być oczywiście używane do dużo bardziej skomplikowanych zadań, ale nawet najbardziej złożone muszą mieć format przedstawiony w tym paragrafie.

## 6.2.2 RETURN

Jak można było zauważyć w deklaracji funkcji, RETURN nie jest użyty w ten sam sposób, co w deklaracjach procedur. W funkcjach po RETURN występuje <wyrażenie arytmetyczne>. W procedurach umieszczenie po RETURN <wyrażenia arytmetycznego> spowoduje wystąpienie błędu.

W funkcjach, podobnie jak w procedurach, RETURN może występować więcej niż jeden raz.

W przykładzie 1 deklaracji funkcji w §6.2.1. obliczany jest kwadrat danej liczby, ale nie ma sprawdzenia czy nie wystąpił nadmiar.

$256 * 256 = 65536$ , a więc o jeden więcej niż jest dopuszczalne dla typu CARD. Są dwa sposoby aby tego uniknąć:

1. liczba wejściowa powinna być typu BYTE, a więc  $\leq 255$
2. sprawdzenie nadmiaru wewnątrz funkcji

**Przykład** poniżej ilustruje drugi z tych sposobów:

```
CARD FUNC square (CARD x)
; ++++ funkcja sprawdza czy x nie jest większe niż 255,
; jeżeli nie, to obliczany jest jego kwadrat, w przeciwnym
; przypadku drukowany jest komunikat o błędzie i za x
; podstawiane jest 0.
IF x >255 THEN
    PrintE ("liczba zbyt duża“)
    RETURN (0)
FI
RETURN (0)
```

**UWAGA:** Jak już wspomniano w §6.1.2 w deklaracji funkcji musi się znajdować przynajmniej jedno RETURN (kompilator tego nie sprawdza).

## 6.2.3 Wywołanie funkcji

Do tej pory spotkaliście się dwukrotnie z wywoływaniem funkcji: w §5.2.4.2 przykład 2 oraz w §5.2.4.3 przykład 1. Były to:

```
num = Rand (101)
guess = InputB()
```

Pierwsze z nich jest przykładem wywołania funkcji, która żąda parametru, drugie jest wywołaniem funkcji bez parametrów. Zarówno funkcja „Rand” oraz „InputB” są funkcjami bibliotecznymi.

„Rand” generuje liczbę z zakresu od 0 do danego w tym przypadku 101 - 1. „InputB” odczytuje wartość z ekranu (klawiatury). Jedna i druga funkcja w wyniku wykonania dostarcza pojedynczą wartość. Ponieważ wartość musi być do czegoś użyta, wywołanie funkcji musi nastąpić w wyrażeniu arytmetycznym. W przykładach powyżej wyrażeniem tym jest instrukcja przypisania.

Wywołanie funkcji może nastąpić w dowolnym, wyrażeniu arytmetycznym z jednym Wyjątkiem:

- wywołania funkcji NIE mogą być używane w wyrażeniach arytmetycznych, jeśli wyrażenie to jest używane jako parametr w wywołaniu lub deklaracji procedury. Np.:

```
x = square (2*Rand (50))
```

Poniżej przedstawiamy kilka poprawnych przykładów wywołania funkcji:

```
x = 5*Rand(201)
c = square(x) - 100
IF ptr <> Peek($8000)
chr = uppercase(chr)
```

„Peek” i „Rand” są funkcjami bibliotecznymi i nie muszą być deklarowane przez użytkownika. „square” i „uppercase” są funkcjami określonymi przez użytkownika i dlatego zanim będą wywołane w programie muszą wcześniej być zadeklarowane.

**UWAGA:** Funkcje można wywoływać w ten sam sposób jak procedury z tym że otrzymana wartość jest wówczas ignorowana.

## 6.3 Zakres zmiennych

Termin zakres zmiennych oznacza granicę obszaru programu, w którym dana zmienna jest zdefiniowana. Aby pomóc Wam zrozumieć co to znaczy użyjemy przykładu z życia.

Poniżej znajduje się tablica ze słowami w języku angielskim i ich synonimami w języku używanym w USA.

British	American
BONNET	HOOD
LORRY	TRUCK
LIFT	ELEVATOR
FAG	CIGARETTE

Każda para słów oznacza to samo, ale obszar językowy, w którym są używane jest inny. „Bonnet” (oznaczający pokrowiec na samochód) jest poprawny tylko w krajach, gdzie używa się czystego języka angielskiego natomiast „Hood” jest poprawny tylko w krajach, gdzie używa się amerykańskiej odmiany tego języka. Tak więc każde z nich ma swój zakres gdzie jest używany. Słowa z lewej kolumny można rozważać jako „globalne” dla języka angielskiego w tym sensie, że każdy przeciętny mieszkaniec obszaru, w którym ten język jest używany, zrozumie co one znaczą. Słowa w prawej kolumnie będą „globalne” dla obszaru językowego, gdzie używa się amerykańskiej odmiany języka angielskiego. W ten oto sposób wytłumaczyliśmy pojęcie zmiennych globalnych. Pora przejść do wyjaśnienia pojęcia zmienna lokalna. Zakres zmiennej lokalnej stanowi pewien podzbiór jakiegoś zakresu globalnego. Jest to analogiczne do tego jak w języku polskim niektóre słowa oznaczają co innego w zależności od sytuacji.

Zmienne w *ACTION!* mają również określony zakres działania. Zakres zmiennej determinuje w jakich częściach programu może być ona użyta.



**Przykład 1:**

```

MODULE
; instrukcja określająca, że zmienne
; zadeklarowane poniżej będą zmiennymi
; globalnymi
CARD numgames =[0],
goal =[10]
beatgoal=[0]
; liczba rozegranych partii
; maksymalna ilość prób
; ilość prawidłowych odgadnięć

PROC intro()
; ++++ procedura ta wyświetla na ekranie zasady gry
CARD ctr
PrintE („Zapraszam do wzięcia udziału w grze.”)
PrintE („Spróbuj odgadnąć liczbę z przedziału 0-100.”)
PrintE („Należy tylko wpisywać swoje odpowiedzi „)
PrintE („kiedy tego zażadam.”)
Put(C)
PrintE („Będę pamiętał ile gier”)
PrintE („rozegrałeś i powiem Ci ile razy”)
PrintE („odgadłeś liczbę w mniejszej ilości”)
PrintE („prób niż to dopuszczalne.”)
PrintE („Podaj mi to maksimum.”)
PutE()
Print („ Wprowadź tutaj maksymalną ilość prób ---> „)
goal = InputC()
FOR ctr = 0 TO 2500 ; pętla czasowa
DO
OD
Put($7D)

RETURN

PROC tally()
; ++++ procedura ta drukuje na ekranie aktualny wynik
Print („Rozegrałeś „)
PrintC (numgames)
PrintE („ partii,“ )
Print („i w “ )
PrintC (beatgoal)
PrintE („ z nich”)
PrintE („odgadłeś liczbę w mniej niż”)
PrintC (goal)
PrintE („ próbach.”)
PutE()

RETURN ; koniec procedury tally

PROC playgame( )
CARD numguesses,
ctr
; ilość prób odgadnięcia liczby
; licznik użyty w pętli czasowej
BYTE num,
guess
; liczba do odgadnięcia
; odpowiedź użytkownika

PrintE („Wybieram swoją liczbę...””)
FOR ctr = 0 TO 4500 ; pętla użyta aby stworzyć wrażenie
DO ; że komputer zastanawia się jaką
OD ; wybrać liczbę

PutE()
PrintE („O.K., możemy zaczynać”)
PutE()
num = Rand (101) ; wygenerowanie liczby, która będzie obgadana
numguesses = 0 ; ustawia na zero licznik prób
DO
Print („Podaj swój typ? „)
guess = InputB() ; wprowadzenie odpowiedzi
numguesses == +1 ; zwiększenie licznika prób o 1

```

```

IF guess < num THEN ; liczba za mała
    PrintE („Zbyt niski, próbuj ponownie „)
ELSEIF guess > num THEN ; liczba za duża
    PrintE („Zbyt wysoki, próbuj ponownie“)
ELSE
    PrintE („Gratulacje!!!!“)
    Print („Odgadłeś w próbie „)
    PrintCE (numguesses)
    IF numguesses < goal THEN
        beatgoal == +1
    FI
FI ; koniec sprawdzania odpowiedzi
UNTIL guess = num ; warunek pętli
OD ; koniec pętli UNTIL
RETURN ; koniec procedury „playgame“
BYTE FUNC stop()
; ++++ funkcja ta sprawdza czy użytkownik chce kontynuować grę
    BYTE again
    PrintE („Chcesz rozegrać „)
    Print („nową partię? /T lub N/ „)
    again = GetD(1) ; wprowadzenie odpowiedzi użytkownika z
                    ; klawiatury w taki sposób aby uniknąć
                    ; wzięcia RETURN za pierwszą odpowiedź
                    ; w nowej partii
    PutE()
    IF again = „N OR again = „n THEN ; zakończenie gry
    RETURN (1)
FI
RETURN (0) ; koniec funkcji /stop/
PROC main( )
    Close (1)
    Open (1, „K:“, 4, 0) ; ustawienie K: tylko na czytanie
    intro() ; drukowanie zasad gry
    DO
    numgames == +1 ; zwiększenie licznika gier
    playgame () ; rozegranie jednej partii
    tally() ; wyświetlenie osiągniętego wyniku w
            ; dotychczas rozegranych partiach
    UNTIL stop() ; koniec pętli jeżeli użytkownik
            ; chce zakończyć grę
    OD
    PutE ()
    PrintE („Spotkajmy się wkrótce.“)
    Close (1)
RETURN ; koniec programu głównego

```



FI

; a dużymi literami alfabetu  
; w kodzie ATASCIT

RETURN (chr)

Należy się teraz zastanowić gdzie zadeklarować „chr”. Wiemy, że można ją zadeklarować jako globalną lub jako lokalną. Jeśli zadeklarujemy ją lokalnie, funkcja musiałaby sama nadawać jej jakąś wartość, a nie o to chodzi. Chcemy aby można było wywołać funkcję „lowertoupper” w następujący sposób (lub podobne):

chr = lowertoupper ( )

Funkcja ma tylko sprawdzać jakie jest „chr” i ewentualnie zamieniać ją na dużą literę alfabetu. Tak więc z pewnością nie możemy zmiennej tej zadeklarować jako lokalną.

Jeżeli „chr” zadeklarujemy globalnie, wówczas osiągniemy to co chcemy ponieważ „chr” w wywołaniu funkcji i wewnątrz funkcji będzie tą samą zmienną. Jest tylko jeden kłopot. Za każdym razem kiedy będziemy chcieli użyć funkcji „lowertoupper”, najpierw trzeba będzie osobną instrukcją ustawić zmienną „chr” np.:

chr = cat  
chr = lowertoupper()  
cat = chr

Jest to uciążliwe jeśli chcemy użyć tej funkcji dla kilku różnych zmiennych. Poza tym jeśli będzie się chciało użyć funkcji „lowertoupper” w innym programie, koniecznie będzie w nim zadeklarowanie również globalnej zmiennej „chr”. Zastosowanie parametrów uwalnia nas od tych kłopotów.

BYTE FUNC lowertoupper (BYTE chr)  
IF chr >= „a AND chr <= „z THEN RETURN (chr-\$20)  
FI  
RETURN

Wywołanie tej funkcji polegać będzie na podstawieniu za parametr zmienną, którą się chce testować. Np.:

chr = lowertoupper (chr)  
cat = lowertoupper(cat)  
var = lowertoupper(var)

Drugą z korzyści jakie się osiąga za pomocą parametrów jest trudniej wytłumaczyć lecz spróbujemy zrobić to w miarę prosto przy użyciu przykładu. Procedura zamieszczona na następnej stronie pobiera dwie liczby typu CARD, dzieli jedną przez drugą i drukuje wynik.

PROC division (CARD num,div)  
num == /div ; zmienia zmienną „num” na iloraz num/div  
PrintC (num^) ; drukowanie wyniku  
RETURN

Przedstawiamy teraz przykład zastosowania procedury „division” w programie.

### Przykład 1:

PROC main()  
CARD ctr,  
number -j\_713j  
FOR ctr = 1 TO 10  
DO  
PrintC (number)  
Print („/“)  
PrintC (ctr)  
Print(“ = “)  
division (number, ctr)  
PutE  
OD  
RETURN

Wynik działania programu 1:

713/1 = 713	713/6 = 118
713/2 = 356	713/7 = 101:
713/3 = 237	713/8 = 89
713/4 = 178	713/9 = 79
713/5 = 142	713/10 = 71

Zwróćcie uwagę, że „number” pozostaje stałe, podczas gdy „num” ulega zmianie. Wartość „number” jest wprowadzana pod „num” przy każdym wywołaniu procedury, lecz po zakończeniu procedury „num” nie jest podstawiana pod „number”. Gdyby zmienna „num” była podstawiana pod „number” wynik byłby następujący:

713/1 = 713	5/6 = 0
713/2 = 356	0/7 = 0
356/3 = 118	0/8 = 0
118/4 = 29	0/9 = 0
29/5 = 5	0/10 = 0

Jak widać przepływ informacji poprzez parametry odbywa się tylko w jednym kierunku; zmienna może być wprowadzona do procedury lecz z reguły nie może być tą drogą wyprowadzona na zewnątrz. Jeżeli chce się na zewnątrz wyprowadzić pojedynczą wartość należy zamiast procedury zastosować funkcję i wartość tą wyprowadzić w instrukcji RETURN. Jeżeli istnieje potrzeba wyprowadzić z procedury większą ilość wartości należy użyć zmiennych globalnych lub jako parametrów zmiennych wskaźnikowych (zob. 9.5).

**UWAGA 1:** Wywołując podprogram z parametrami, pierwszy parametr w wywołaniu jest podstawiany pod pierwszy parametr deklaracji podprogramu, drugi za drugi itd. Przy wywołaniu podprogramu można podać mniej parametrów niż jest to zadeklarowane, ale nie więcej.

Np. jeżeli w deklaracji jest 5 parametrów, można wywołać ten podprogram z 0–5 parametrami. Pozwala to pisać podprogramy, które wymagają zmiennej ilości parametrów zależnie od zadania, które mają wykonać.

**UWAGA 2:** Kompilator nie wykrywa błędu jeżeli przy wywołaniu podprogramu użyje się parametru innego typu niż jest zadeklarowany. Jeżeli użyje się typu CARD podczas gdy procedura wymaga zmiennej typu BYTE, pod zmienną zostanie podstawiony bajt LSB typu CARD.

**UWAGA 3:** Parametrami podprogramów mogą być:

1. zmienne podstawowego typu danych
2. element tablicy, wskaźnik lub pole rekordu
3. nazwy tablicy zmiennej wskaźnikowej lub rekordu

W przypadku trzecim nazwy są użyte jako wskaźnik do pierwszego elementu, wartość lub pierwsze pole w nazwanej zmiennej.

## 6.5 MODULE

MODULE jest dyrektywą bardzo prostą. Jej format jest następujący:

```
MODULE
```

Dyrektywa ta sygnalizuje kompilatorowi, że użytkownik chce zadeklarować pewną ilość zmiennych globalnych. Jest to użyteczne gdy pisze się duży program częściami, które zawierają swoje zmienne globalne. Jeżeli umieści się MODULE na początku każdej części kompilator dołączy dane zmienne globalne do tak zwanej tablicy zmiennych globalnych.

Nie jest konieczne zastosowanie w programie dyrektywy MODULE, ponieważ kompilator automatycznie przyjmuje, że zmienne umieszczone na początku programu są zmiennymi globalnymi. Tak więc deklaracja zmiennych globalnych może następować albo na początku programu, albo bezpośrednio po dyrektywie MODULE.

## Rozdział 7 Dyrektywy kompilatora

Dyrektywy kompilatora różnią się od zwykłych instrukcji języka tym, że są one wykonywane podczas kompilacji a nie po uruchomieniu programu.

### 7.1 DEFINE

Dyrektywa ta jest podobna do komendy edytora (<CS>S), z tą różnicą, że jest ona wykonywana podczas kompilacji. Format:

```
DEFINE <identyfikator> = <stała tekstowa> {,<identyfikator> = <stała tekst>}
```

gdzie: <stała tekstowa> jest łańcuchem tekstowym wraz z cudzysłowami.

Dyrektywa ta jest używana do poprawienia czytelności programu, kompilator w każdym miejscu programu gdzie jest użyty dany identyfikator podstawia w jego miejsce wyspecyfikowany tekst. Np. jeżeli jest kompilowany program zawierający linię:

```
DEFINE size = „256”
```

we wszystkich miejscach, gdzie występowało „size” zostanie podstawione 256. Pozwala to na wiele interesujących rozwiązań. Jeżeli z pewnych względów nie lubisz używać słowa kluczowego CARD, możesz je zmienić np. na „PRÓG” używając komendy:

```
DEFINE PROG = „CARD”
```

W ten sposób możesz napisać program używając nazwy „PROG” a kompilator i tak będzie je traktował jako „CARD”.

#### Przykłady:

```
DEFINE liston = „SET $49A=1”
DEFINE begin = „DO”,end = „OD”
DEFINE one = „1”
```

**UWAGA:** Stała tekstowa musi być zawsze ujęta w cudzysłowy.

**Przykład** poniżej ilustruje efekt działania dyrektywy DEFINE.

```
DEFINE four = „4”           ; dyrektywa
PrintBE (four)              ; drukuje „4” .
; four score and            ; zmienia „four” na „4”
; four-score and            ; „four-score” pozostaje bez zmian
Prints („four score”)       ; nie zmienia nazw ujętych w cudzysłowy
```

### 7.2 INCLUDE

Dyrektywa INCLUDE pozwala wstawić w kompilowany program inne programy. Załóżmy, że mamy program nazwany „IOSTUFF.ACT”, który wykonuje operacje wejścia/wyjścia i chcemy użyć podprogramów, które on oferuje w swoim, aktualnie pisanym programie. Do programu, który jest pisany, należy wprowadzić następującą linię:

```
INCLUDE „D1:IOSTUFF.ACT”
```

**UWAGA:** Specyfikacja pliku musi być ujęta w cudzysłowy.

Powyższa instrukcja musi wystąpić wcześniej niż nastąpi użycie podprogramów z pliku „IOSTUFF.ACT”. Jeżeli urządzenie, z którego ma być odczytany plik nie jest wyspecyfikowane, kompilator przyjmie automatycznie, że jest to „D1:”. Plik może być ściągnięty do pamięci z dowolnego urządzenia, z którego jest dozwolony odczyt („P:” nie jest poprawne).

#### Przykłady:

```
INCLUDE „D2:IOLID.ACT”
INCLUDE „PRG1.DAT”
INCLUDE „C:”
```

**UWAGA:** Większość systemów operacyjnych wymaga aby specyfikacja pliku była napisana dużymi literami alfabetu.

Za pomocą komendy `INCLUDE` można wstawić program, który również za pomocą tej komendy wstawia w swój tekst inne programy (efekt zagnieżdżenia). W *ACTION!* dopuszcza się zagnieżdżenia do 6 poziomów lecz urządzenia peryferyjne oraz system operacyjny mają inne ograniczenia. Jeżeli przekroczy się ograniczenia systemu operacyjnego, pojawi się błąd 161 (zbyt dużo otwartych plików). Dla pamięci kasetowej ograniczenie to wynosi 1, a dla napędu dyskowego 3. Jeżeli aktualniej buforze edytora *ACTION!* nie znajduje się żaden program, maksymalna liczba poziomów komendy `INCLUDE` jest zmniejszona o jeden.

## 7.3 SET

Dyrektywa `SET` jest używana do zmodyfikowania zawartości pamięci RAM (*Random Access Memory*). `SET` podczas procesu kompilacji programu ładuje do określonej komórki pamięci nową wartość. W większości przypadków komenda ta jest używana do zmiany opcji edytora i kompilatora z poziomu programu użytkownika. Nie może być natomiast, użyta do modyfikacji programu, systemu operacyjnego lub zmiennych sprzętowych.

Format:

```
SET <adresy> = <wartość>
```

**UWAGA:** <adres> i <wartość> muszą być stałymi kompilacji.

Jeżeli <wartość> jest większa niż 255, jest wstawiana pod komórkę <adres> i <adres+1>. Dzieje się tak dlatego, że 255 jest największą liczbą, która się mieści w pojedynczym bajcie.

**Przykłady:**

```
SET $600 = 64           ; komórka o adresie $600 przyjmuje wartość 64
SET max = 16           ; wstawia pod „max” wartość 16
SET 10000 = $FFFF     ; wstawia do komórek 10000 i 10001 wartość $FFFF
SET $CF00 = cad       ; wstawia do komórek $CF00 i $CF01 wartość @cad
DEFINE add = „$7000”
SET add = $42
```

Przykład ostatni pokazuje stałą numeryczną zdefiniowaną przez `DEFINE`, użytą w instrukcji `SET`. Ponieważ zmienne w dyrektywie `DEFINE` są stałymi kompilacji, można je użyć w dyrektywie `SET`, ale tylko w tej kolejności jak w powyższym przykładzie (najpierw dyrektywa `DEFINE`, później `SET`).

**UWAGA:** nie należy mylić komendy `SET` z dającymi podobny efekt, ale w czasie działania programu, instrukcjami `Poke` i `PokeC`.

## Rozdział 8 Złożone typy danych

Złożone typy danych powodują, że *ACTION!* jest bardziej elastyczny niż większość języków programowania dostępnych na ATARI.

W skład tych typów wchodzi:

1. zmienne wskaźnikowe
2. tablice
3. rekordy

### 8.1 Zmienne wskaźnikowe

Wskaźnik kojarzy się z przyrządem służącym nauczycielowi do pokazywania określonych miejsc na mapie. W *ACTION!* wskaźnik oznacza coś bardzo podobnego.

Zmienne wskaźnikowe zawierają adres pamięci, a więc wskazują na określoną komórkę pamięci. Poprzez zmianę wartości takiej zmiennej uzyskuje się np., że wskazuje ona nowe miejsce (tak samo jak nauczyciel przesuwa wskaźnik na inne miejsce na mapie). Różnica jest taka, że wskaźnik nauczyciela wskazuje miasta lub rzeki, a zmienne wskaźnikowe w *ACTION!* wartości typu `BYTE`, `CARD` lub `INT`. Kompilator musi być poinformowany jakiego typu wartości będzie wskazywać dana zmienna wskaźnikowa.

Najpierw, wobec tego, omówimy sposób deklarowania zmiennych wskaźnikowych, następnie pokażemy za pomocą programów sposób ich użycia.

#### 8.1.1 Deklaracja zmiennych wskaźnikowych

Format:

```
<typ> POINTER <identyfikator>{=<adres>} | : ,<identyfikator>{=<adres>}:|
```

gdzie:

<code>&lt;typ&gt;</code>	;	podstawowy typ danych. Określa jakiego typu wartości wskazuje dana zmienna wskaźnikowa
<code>POINTER</code>	;	słowo kluczowe określające deklarację zmiennych wskaźnikowych
<code>&lt;identyfikator&gt;</code>	;	nazwa zmiennej
<code>&lt;adres&gt;</code>	;	opcjonalnie ustawia zmienną wskaźnikową na adres początkowy. <code>&lt;adres&gt;</code> musi być stałą kompilacji.

Ponieważ zmienna wskaźnikowa zawiera adres, musi być ona liczbą z przedziału 0-65535 (\$0000-\$FFFF) ponieważ ATARI ma 64 KB pamięci.

Tak więc zmienne wskaźnikowe są zapamiętywane na dwóch bajtach (LSB, MSB) w typie `CARD`.

Zastosowanie zmiennych wskaźnikowych zilustrowane będzie w następnym paragrafie. Poniżej pokazujemy tylko kilka przykładów deklaracji tych zmiennych:

```
BITE POINTER ptr          ; deklaruje ptr jako wskaźnik dla wartości typu BYTE
CARD POINTER cpl          ; deklaruje cpl jako wskaźnik do wartości typu CARD
INT POINTER ipe=$8000     ; deklaruje ipe jako wskaźnik do wartości typu INT i inicjuje tę zmienną
                           aby wskazywała adres $8000
```

#### 8.1.2 Operacje na zmiennych wskaźnikowych

Zmienne wskaźnikowe mogą być szeroko stosowane w *ACTION!* ponieważ można nimi łatwo manipulować tak, aby wskazywały różne komórki pamięci. Pozwala to w prosty sposób budować katalogi, listy danych itd.

Program poniżej jest prostym przykładem na to, co można osiągnąć za pomocą wskaźników. Operator „^” w instrukcji przypisania oznacza, że w miejsce, które wskazuje zmienna wskaźnikowa należy umieścić żadaną wartość.

```
PROC pointerusage ()
    BYTE num = $E0,          ; deklaracja i umieszczenie pod zadanymi
```



```

chr = $E1 ; adresami dwóch zmiennych typu BYTE
BYTE POINTER bptr ; deklaracja zmiennej wskaźnikowej do
; wartości typu BYTE

bptr =@num ; powoduje, że bptr będzie wskaźnikiem do
Print(„bptr jest obecnie wskaźnikiem do adresu”)
Printf(„%H”, bptr) ; drukuje adres zmiennej num
PutE()
Bptr^ = 255 ; umieszcza 255 w miejscu, które wskazuje
; wskaźnik (tzn.w num)

Print(„num wynosi obecnie :)
PrintBE(num) ; pokazuje, że 255 zostało rzeczywiście
; umieszczone w zmiennej num
; umieszcza 0 w zmiennej num

Bptrt^ = 0
Print(„num wynosi obecnie „)
PrintBE(num) ; pokazuje, że num równa się teraz 0
bptr =@chr ; powoduje, że bptr staje się wskaźnikiem
; do chr

Print(„bptr jest obecnie wskaźnikiem do adresu „)
Printf(„%H”, bptr) ; drukuje adres zmiennej chr, tak abyśmy
; wiedzieli, że bptr rzeczywiście się zmieniło
bptr* = „q ; umieszcza „q w miejscu określonym
; przez wskaźnik (tzn. w chr)

Print(„chr wynosi obecnie “)
Put(chr) ; pokazuje, że chr równa się „q
PutE()
Bptr^ = „z ; zmiana zawartości chr na „z
Print(„chr wynosi obecnie” „)
Put(chr) ; pokazuje, że chr równa się „z
PutE( )
RETURN

```

### Wynik działania programu 1:

```

bptr jest obecnie wskaźnikiem do adresu $EO
num wynosi obecnie 255
num wynosi obecnie 0
bptr jest obecnie wskaźnikiem do adresu $E1
chr wynosi obecnie q
chr wynosi obecnie z

```

Zauważcie, że operator „^” został użyty do umieszczenia określonej wartości w miejsce, określone przez zmienną wskaźnikową. Tak więc, linia „bptr^=0” w powyższym przykładzie jest tym samym, co „num=0”, ponieważ „bptr” wskazywało wtedy zmienną „num”.

Zmienne wskaźnikowe mogą być użyte w wyrażeniu arytmetycznym np.

```
x = ptr^
```

Instrukcja „Printf(„%H”, bptr)” jest poprawna ponieważ „bptr” może być traktowany równie dobrze jako liczba. Jest to użyteczne podczas testowania programu, ponieważ można w prosty sposób wyprowadzić na zewnątrz adres, na który jest w danym momencie ustawiony wskaźnik.

## 8.2 Tablice

Tablice są najbardziej znanymi złożonymi strukturami danych. Pozwalają manipulować całym ciągiem zmiennych używając do tego tylko nazwy tablicy oraz indeksu. Elementami tablicy są zmienne jednego typu (typu podstawowego). Nazwa tablicy określa tablicę, na której są wykonywane operacje, natomiast indeks jest liczbą, która wskazuje na konkretny element tablicy.

## 8.2.1. Deklaracja tablicy

Format:

```
ARRAY <def. tablicy> |:,<def. tablicy>:|
```

gdzie: <typ> ; określa typ elementów tablicy /musi to być typ podstawowy/  
 ARRAY ; słowo kluczowe oznaczające deklarację tablicy  
 <def.tablicy> ; jest informacją potrzebną do zadeklarowania zmiennej jako tablicy o elementach określonego typu.

<def.tablicy> ma następujący format:

```
<identyfikator>{(wymiar)}{=<adres>|[<wartości>]|<stała tekstowa>}
```

gdzie: <identyfikator> ; jest nazwą zmiennej  
 <wymiar> ; jest wymiarem tablicy i musi być stałą numeryczną  
 <adres> ; jest adresem pierwszego elementu tablicy i musi być stałą kompilacji  
 <wartości> ; ustawia wartości początkowe elementów tablicy. Muszą być stałymi numerycznymi  
 <stała tekstowa> ; ustawia wartości początkowe elementów tablicy. W pierwszym elemencie tablicy zostaje umieszczona długość tego tekstu, w pozostałych tekst.

### Przykłady:

```
BYTE ARRAY a,b ; deklaracja dwóch tablic z elementami typu BYTE. Wymiar nie jest zdefiniowany
INT ARRAY x(10) ; deklaracja „x” jako tablicy 10-cio elementowej
BYTE ARRAY str = „To jest stała tekstowa.” ; deklaracja „str” jako tablicy typu BYTE. Tablica ta zostanie wypełniona tekstem
CARD ARRAY junk = $8000 ; deklaracja „junk” jako tablicy typu CARD, której początek będzie się znajdował pod adresem $8000. Wymiar nie jest podany
BYTE ARRAY tests = [4 7 18] ; deklaracja „tests” jako tablicy typu BYTE i wypełnienie jej wartościami podanymi w nawiasach
```

**UWAGA:** Wymiar tablicy powinno się z reguły podawać jeżeli jest to tylko możliwe. Istnieje jednak kilka sytuacji kiedy nie trzeba lub nie można tego robić:

1. kiedy nie wiadomo jak duża będzie tablica (np. jeżeli jest parametrem podprogramu i nie wiadomo jak duża tablica będzie podstawiana przy wywołaniu).
2. jeżeli w deklaracji tablicy wypełnia się ją od razu wartościami (używając konstrukcji „[<wartości>]” lub <stała tekst.>,) i nie planuje się zwiększanie tej tablicy.

Należy pamiętać, że pierwszy bajt stałej tekstowej zawiera jej długość. Tak więc, aby wydłużyć łańcuch najpierw należy zmienić bajt długości (który jest zerowym elementem tablicy zawierającej ten łańcuch).

## 8.2.2. Reprezentacja wewnętrzna tablic

Reprezentacja wewnętrzna tablic jest bardzo podobna do wskaźników.

Nazwa tablicy jest wskaźnikiem do jej pierwszego elementu. Cała tablica jest ciągiem „komórek”, z których każda zawiera jeden jej element. Rozmiar takiej „komórki” jest zdeterminowany typem elementów tablicy: jeden bajt dla typu BYTE, dwa bajty dla typów CARD i INT.

## 8.2.3 Korzystanie z tablic

Stosowanie tablic jest proste jeżeli wie się tylko jak je zadeklarować i jak odwołać się do poszczególnych elementów. Ilustrują to poniższe przykłady.

### Przykład 1:

```
PROC reftest()
  BYTE x
  BYTE ARRAY nums (10)
```

```

FOR x = 0 TO 9          ; nums jest tablicą 10-cio elementową indeks zmienia się od 0 do 9,
                        ; a nie od 1 do 10
DO
  nums(x) = x+ „A     ; do elementu o indeksie x jest przypisana wartość x+”A
  Put vnuras (x)      ; wydrukowanie elementu o indeksie x w postaci znakowej
  Print(„ „)         ; pozostawienie wolnego miejsca pomiędzy drukowanymi znakami
OD
PutE()
RETURN

```

Wynik działania programu 1:

A B C D E F G H I J

W programie użyto dwóch odwołań do pojedynczych elementów tablicy: `nums(x)` w instrukcji przypisania oraz „`nums(x)`” jako parametr procedury bibliotecznej „`Put`”. Te i wszystkie inne odwołania do konkretnego elementu tablicy mają postać:

<identyfikator>(<indeks >)

gdzie: <identyfikator> jest nazwą tablicy, do której następuje odwołanie  
 <indeks> jest indeksem konkretnego elementu tablicy i jest to wyrażenie arytmetyczne

Pierwszy element tablicy ma indeks 0, a nie jak można by tego oczekiwać 1.

**Przykład 2:**

```

PROC cchangearray()
  BYTE ARRAY barray
  barray = „to jest łańcuch 1.”
  PrintC(barray)          ; drukuje adres „barray” jako wartość typu CARD
  Print(„ „)
  PrintE(barray)         ; drukuje łańcuch barray jako tekst (łącznie ze znakiem
                          ; końca linii EOL)

  barray = „to jest łańcuch 2.”
  PrintC(barray)
  Print(„ „)
  PrintE(barray)
RETURN

```

Wynik działania programu 2:

10352 to jest łańcuch 1.  
 10414 to jest łańcuch 2.

**KOMENTARZ DO PRZYKŁADU 2:** Z wydruku można zobaczyć, że adres, na który wskazuje „`barray`” ulega zmianie. Ponowne przypisanie do całej tablicy nowej wartości (przy użyciu stałej tekstowej) powoduje, że nowy łańcuch jest umieszczany w innym miejscu pamięci niż poprzedni. Dlatego też adres początkowy tablicy automatycznie ulega zmianie. Poprzedni tekst znajduje się nadal w pamięci, ale nie ma do niego dostępu.

**Przykład 3:**

```

PROC equatearrays()
  BYTE ARRAY a = „To jest stała tekstowa.”,
              barray
  barray = a
  PrintE(a)
  PrintE(barray)
RETURN

```

Wynik działania programu 3:

To jest stała tekstowa.  
 To jest stała tekstowa.

Jak widać z tego przykładu istnieje bardzo prosty sposób przypisania wartości jednej tablicy do drugiej. Wystarczy ustawić je w taki sposób żeby wskazywały ten sam adres pamięci.

**Przykład 3:**

```
BYTE ARRAY a = [„T „e „k „s „t]
PrintE(a)
```

Program powyższy nie będzie działał poprawnie. Należy pamiętać, że stałe tekstowe są czymś innym niż zwykły tekst (łańcuch), ponieważ ich pierwszy bajt zawiera długość łańcucha. Procedura PrintE wymaga jako parametru stałej tekstowej, a nie tekstu. Tak więc wywołanie tej procedury w programie jest niewłaściwe (tablica zawiera tekst, a nie stałą tekstową).

**Przykład 4:**

Załóżmy, że masz program, który podaje numer błędu popełnionego przez użytkownika. Chcesz aby oprócz tego był drukowany komunikat o rodzaju błędu. Można do tego użyć tablicy.

```
PROC doerror (BYTE ermum)
;++++ procedura ta odczytuje numer błędu, a następnie, drukuje
; informację co to za błąd.
    BYTE ARRAY errmsg                ; komunikat wyprowadzany na ekran
    CARD ARRAY addr(6)                ; przechowuje adresy poszczególnych komunikatów
    addr(0)=„Nielegalna komenda.”
    addr(1)=„Nielegalny znak.”
    addr(2)=„Błędna nazwa pliku.”
    addr(3)=„Liczba zbyt duża.”
    addr(4)=„Niepoprawny typ liczby.”
    addr(5)=„Błąd nierozpoznany.”
    eermsg = addr(ermum)                ; umieszcza komunikat o błędzie
    Print („BŁĄD = „)                  ; w tablicy „errmsg”. „ermum” zawiera numer błędu
    PrintB(ermum)
    Print („: “)
    PrintE(errmsg)
    PutE( )
RETURN                                ;++++ koniec procedury „doerror”

PROC main()
;++++ procedura ta jest sztucznie użyta do wywołania procedury
; „doerror” używając wszystkich możliwych numerów błędów.
    BYTE error
    FOR error = 0 TO 5
        DO
            doerror(error)
        OD
RETURN                                ;++++ koniec procedury „main”
```

**Wynik działania programu 4:**

```
Błąd = 0: Nielegalna komenda.
Błąd = 1: Nielegalny znak.
Błąd = 2: Błędna nazwa pliku.
Błąd a 3: Liczba zbyt duża.
Błąd = 4: Niepoprawny typ liczby.
Błąd = 5: Błąd nierozpoznany.
```

**KOMENTARZ DO PRZYKŁADU 4:** Sposób w jaki w powyższym przykładzie została zapełniona tablica typu CARD jest dosyć niezwykły, ale jednak poprawny. Do poszczególnych elementów tablicy nie są przypisywane całe stałe tekstowe, a tylko ich adresy. Dzięki temu, każdy element tablicy staje się wskaźnikiem do określonego łańcucha. Następnie wystarczyło już tylko przypisać wartość określonego elementu tablicy do tablicy „errmsg” typu BYTE i „errmsg” stało się wskaźnikiem do określonego komunikatu i można go było wydrukować.

Program 4 może wyglądać na dosyć skomplikowany jeżeli w pełni nie rozumiecie idei tablic i ich wewnętrznej reprezentacji.

## 8.3 Rekordy

Rekordy są konstrukcjami pozwalającymi grupować informacje, które chociaż są ze sobą w jakiś sposób powiązane, nie są tego samego typu. Przykładem takiego rekordu jest zestaw informacji zapisany w dowodzie osobistym. Oprócz nazwiska i imion, które są tekstem, występują między innymi takie informacje jak wzrost, kolor oczu, data urodzenia oraz adres zamieszkania. Jak można zauważyć są one różnych typów. Oczywiście, w *ACTION!* nie mogą wszystkie z nich wystąpić. Rekordy grupują tylko te informacje, które może zrozumieć kompilator. Muszą one należeć do zbioru podstawowych typów danych.

### 8.3.1 Deklaracja rekordów

§8.3.1.1. pokazuje jak stworzyć typy rekordowe, natomiast §8.3.1.2. demonstruje jak zadeklarować zmienne tego typu.

#### 8.3.1.1. Deklaracja TYPE

Format:

```
TYPE <identyfikator>=[<deklaracje zmiennych>]
```

gdzie:	TYPE	słowo kluczowe oznaczające typ rekordowy
	<identyfikator>	jest nazwą tego typu
	<deklaracje zmiennych>	są zwykłymi deklaracjami zmiennych, takimi jak w §3.4.1 z wyjątkiem tego, że opcja „=<wart.pocz.>” nie jest dozwolona

**Przykłady:**

```
TYPE rec = [BYTE b1,b2           ; pierwsze dwa pola będą typu BYTE
            INT i1               ; trzecie pole typu INT
            CARD c1,c2,c3       ; następne trzy typu CARD
            BYTE b3]           ; ostatnie pole typu BYTE
```

Przykład ten wymaga wytłumaczenia więc będziemy go analizować krok po kroku:

TYPE rec	; definiujemy nowy typ danych nazwany „rec”
BYTE b1,b2	; pierwsze dwa pola typu rec będą typu BYTE i są nazwane „b1” i „b2”
INT i1	; trzecie pole będzie typu INT i jego nazwa brzmi „i1”
CARD c1,c2,c3	; pola czwarte do szóstego będą typu CARD i są nazwane odpowiednio jako „c1”, „c2” i „c3”
BYTE b3	; siódme pole rekordu „rec” (ostanie) będzie typu BYTE i zostało nazwane „b3”

Należy zwrócić uwagę, że pomiędzy deklaracjami imiennych różnych typów nie występują przecinki. Gdyby je tam umieścić, kompilator próbowałby odczytać słowa kluczowe oznaczające typ danych (BYTE, CARD, INT) jako zmienne, co spowodowałoby wystąpienie błędu.

#### 8.3.1.2 Deklaracja zmiennych rekordowych

W paragrafie tym pokażemy w jaki sposób zadeklarować zmienne danego typu rekordowego. Format:

```
<identyfikator><zmienna>{=<adres>}|:<zmienna>{=<adres>}|
```

gdzie:	<identyfikator>	jest nazwą typu rekordowego
	<zmienna>	jest zmienną, którą chcemy zadeklarować jako danego typu rekordowego
	<adres>	jest adresem komórki pamięci, w której chcemy aby dana zmienna została ulokowana. Musi to być stała numeryczna

W poniższym przykładzie użyjemy typu rekordowego zadeklarowanego w poprzednim paragrafie.

```
TYPE rec = [BYTE b1,b2
            INT i1
            CARD d,c2,c3
            BYTE b3 3
```

```

rec avec,                ; deklaracja avec jako typ rekordowy rec
    brec = $8000         ; deklaracja brec jako typ rekordowy rec i umieszczenie jej
                        ; pod adresem $8000

```

**KOMENTARZ:**

```

rec                ; wskazuje, że zmienne zapisane po tym słowie będą typu rekordowego „rec”, podobnie
                  ; jak to było ze słowami BYTE, INT i CARD
avec              ; zmienna „avec” będzie typu rekordowego „rec”
brec = $8000     ; deklaruje „brec” jako zmienną typu „rec” i umieszcza ją pod adresem $8000

```

### 8.3.2 Sposób użycia rekordów

Aby dostać się do określonego pola rekordu należy użyć operatora („.”).

**Przykład 1 :**

```

PROC recordreference()
;++++ program odczytuje pewne informacje o pracowniku, a następnie
; drukuje je, aby można było sprawdzić czy są poprawne
    TYPE idinfo = [BYTE level,                ; poziom zatrudnienia
                  CARD idnum,                ; numer identyfikacyjny
                  entry_year]                ; rok podjęcia pracy
    idinfo rec                                ; deklaracja rec jako typ rekordowy
idinfo
Print („Jaki jest Twój numer identyfikacyjny?”)
rec.idnum = InputC()                          ; wprowadzenie tego numeru
Print („Jaki jest Twój poziom zatrudnienia /A-Z/? „)
rec.level = GetD(7)                            ; wprowadzenie tego poziomu
Print („W którym roku rozpoczęłeś pracę? „)
rec.entry_year = InputC()                      ; wprowadzenie roku podjęcia pracy
PrintE („O.K. Sprawdź czy się wszystko zgadza:”)
PutE()
Print („Numer = „ )                          ; drukowanie
PrintCE (rec.idnum)                           ; informacji,
Print („Poziom: „)                            ; które
Put (rec.level)                               ; wprowadził
PutE()                                         ; pracownik
Print („Rok zatrudnienia:”)
PrmtCE(rec.entry_year)
RETURN

```

**Wynik działania programu 1:**

```

Jaki jest Twój numer identyfikacyjny? 4565
Jaki jest Twój poziom zatrudnienia /A-Z/? L
W którym roku rozpoczęłeś pracę? 1979
O.K. Sprawdź czy się wszystko zgadza:
Numer = 4365
Poziom: L
Rok zatrudnienia: 1979

```

Znak „.” jest użyty do odwołania się do konkretnego pola rekordu.

Z przykładu widać, że odwołanie się do jednego pola rekordu ma następującą postać:

```
<nazwa rekordu>.<nazwa pola>
```

Należy zwrócić uwagę, że <nazwa pola> i <nazwa rekordu> są zdefiniowane w odrębnych deklaracjach.

## 8.4 Dodatkowe rady odnośnie stosowania złożonych typów danych

Wydaje się, że złożone typy danych są ograniczone tym, że mogą tylko działać na typach podstawowych. Nie można użyć rekordów w tablicach, tablicy jako pola rekordu itd. Istnieją jednak sposoby aby obejść te ograniczenia tak jak to zrobiono w przykładzie 4 w §8.2.3.

W przykładzie tym została utworzona tablica, której poszczególne elementy były zmiennymi wskaźnikowymi, a nie zwykłymi liczbami. W tym paragrafie zademonstrujemy inne sposoby pozwalające na uzyskanie bardziej złożonych typów danych. Zademonstrujemy program, w którym rekordy mają pola będące tablicami oraz program, który stosuje tablice zbudowane z rekordów. Zobaczycie, że typy takie są nielegalne tylko wtedy jeżeli próbuje się to zrobić wprost.

Przykład poniżej zapełnia tablicę ciągiem rekordów. Nie będą to rekordy w tym sensie, że będą zadeklarowane jako typ rekordowy, ale będzie to zwykły ciąg bajtów umieszczonych w pewnym miejscu pamięci. Zadeklarujemy typ rekordowy i jednocześnie wskaźnik do tego typu. Tablicą będziemy manipulować w ten sposób, że będzie ona „podzieloną” na bloki o rozmiarze jednego rekordu i będzie utworzony wskaźnik, który pozwoli przeskakiwać od bloku do bloku.

### Przykład 1:

```

MODULE                                ; deklaracja zmiennych globalnych
    TYPE idinfo=[CARD idnum,          ; numer identyfikacyjny pracownika
                  codenum             ; jego kod dostępu
                  BYTE level]        ; poziom zatrudnienia
    BYTE ARRAY idarray (1000)        ; wymiar pozwalający na przechowanie 200 rekordów
    DEFINE recordsize = „5”
    CARD reccount = [0]
PROC fillinfo()
;++++ procedura ta pobiera informację o danym pracowniku, umieszcza ją
; w tablicy rekordów używając wskaźnika do typu rekordowego.
; Proces ten jest kontynuowany tak długo jak to jest potrzebne.
    idinfo POINTER newrecord
    BYTE continue
    DO
        newrecord = idarray + (reccount * recordsize)
        Print („Numer identyfikacyjny? „)
        newrecord.idnum = InputC()
        Print („Poziom zatrudnienia /A-Z/? „)
        newrecord.level = GetD(7)
        Print („Kod dostępu? „)
        newrecord.codenum = InputC()
        reccount == +1
        PutE()
        Print („Wprowadzasz następny rekord /T lub N/? „)
        continue = GetD(7)
        PutE()
    UNTIL continue = „N OR continue = „n
    OD
RETURN

```

**UWAGA:** Procedura ta nie sprawdza czy nastąpiło wyjście poza granicę tablicy.

### KOMENTARZ DO PRZYKŁADU 1:

```
DEFINE recordsize = „5”
```

Instrukcja ta określa wielkość „skoku” podczas poruszania się w tablicy. Rekord typu „idinfo” ma 5 bajtów długości (2 typu CARD i 1 typu BYTE). Aby wyeliminować możliwość zapisania jednego rekordu na części drugiego, za każdym razem przesuwamy się o 5 bajtów.

```
idinfo POINTER newrecord
```

Jest to deklaracja wskaźnika do typu „idinfo”. Wskazuje on pierwszy wolny element tablicy, na którym może być zapisane pierwsze pole rekordu.

```
newrecord = idarray + (reccount * recordsize)
```

Instrukcja ta powoduje, że zmienna wskaźnikowa pokazuje na pierwszy wolny element tablicy. Dokonuje się tego przez dodanie do początkowego adresu tablicy obszaru, jaki zajmują wszystkie do tej pory wprowadzone rekordy (ilość rekordów /reccount/ pomnożona przez rozmiar jednego rekordu /recordsize/).

```
newrecord.xxx = XXX
```

„newrecord” jest nazwą pola rekordu, a „xxx” odpowiada funkcji wejścia/wyjścia użytej do wypełnienia tablicy.

```
reccount == +1
```

Zwiększenie o jeden wartości zmiennej, która zlicza liczbę rekordów znajdujących się aktualnie w tablicy.

Spróbujmy dodać do rekordu jedno pole więcej zawierające imię i nazwisko pracownika. W definicji rekordu dodamy jedno pole i zmienimy dyrektywę DEFINE ponieważ długość rekordu liczona w bajtach ulegnie zmianie. Zwiększymy ją o 20. W ten sposób tekst będzie mógł mieć długość 13 bajtów ponieważ pierwszy bajt przeznaczony jest na długość tego tekstu.

### Przykład 2:

```
MODULE
    TYPE ininfo =[CARD idnum,
                  Codenum
                  BYTE level,
                  Name]
    BYTE ARRAY idarray(1000)      ; wymiar pozwalający na przechowanie 40 rekordów
    DEFINE recordsize = „25”
        nameoffset = „5”
    CARD reccount = [0]
PROC fillinfo()
;++++ jest to zmodyfikowana wersja procedury z poprzedniego przykładu
    idinfo POINTER newrecord
    BYTE POINTER nameptr          ; wskaźnik do pola „name”
    BYTE continue
    DO
        newrecord = idarray + (reccount * recordsize)
        Print(„Numer identyfikacyjny? „)
        newrecord.idnum = InputC()
        Print(„Poziom zatrudnienia /A-Z/? „)
        newrecord.level = GetD(7)
        Print(„Kod dostępu? „)
        newrecord.codenum = InputC()
        nameptr = newrecord + nameoffset      ; ustawienie „nameptr” na początek pola
                                                „name”
        Print(„Imię i nazwisko pracownika?“)
        InputS(nameptr)
        reccount == +1
        PutE()
        Print(„Wprowadzasz następny rekord /T lub N/? „)
        continue = GetD(7)
        PutE()
    UNTIL continue = „N OR continue =„n
    OD
RETURN
```

### Omówienie przykładu 2

```
nameoffset = „5”
```

Instrukcja ta definiuje ilość bajtów jaką trzeba ominąć w rekordzie aby dostać się do pierwszego bajtu przeznaczonego na tekst.

```
BYTE POINTER nameptr
```

Zmienna ta jest używana do wskazania w rekordzie pierwszego bajtu pola „name”.

```
nameptr = newrecord + nameoffset
```

Ustawianie wartości wskaźnika „nameptr” przez dodanie do początkowego adresu rekordu („newrecord”) liczby bajtów jaką trzeba opuścić, aby się dostać do pierwszego bajtu tekstu.



InputS(nameptr)

Instrukcja ta jest użyta do wprowadzenia imienia i nazwiska pracownika. Wykorzystuje się w tym celu „nameptr” jako wskaźnik, w którym miejscu pamięci mają być one umieszczone. Podobna operacja miała miejsce w przykładzie 2 §8.2.3, z tym że zamiast wskaźnika była tam użyta nazwa tablicy, która jest przecież niczym innym jak wskaźnikiem do pierwszego elementu tablicy.

Obecnie, wiedząc już jak umieszczać rekordy w tablicy, pokażemy jak odnaleźć w tablicy rekordów zadany rekord. Użyjemy do tego funkcji, a nie procedury ponieważ będziemy chcieli uzyskać tylko pojedynczą wartość: adres do pierwszego rekordu z polem idnum równym zadanej wartości. Jeżeli rekord taki nie zostanie odnaleziony wartość funkcji będzie wynosiła 0 (zero). Funkcja ta używa zmiennych globalnych z poprzedniego przykładu.

### Przykład 3:

```
CARD FUNC findmatch (CARD testidnum)
    idinfo POINTER seeker                ; wskaźnik do testowanego rekordu
    BYTE ctr                              ; licznik pętli FOR
    FOR ctr = 0 TO (reccount-1)          ; minus jeden ponieważ rozpoczynamy
    DO                                    ; od 0, a nie od 1
        seeker=idarray+(ctr*recordsize) ; indeks rekordu
        IF seeker.idnum = testidnum THEN ; sprawdzanie czy pole idnum
            RETURN (seeker)              ; danego rekordu jest równe zadanej
        FI                                ; wartości i jeżeli tak to koniec
    OD                                    ; wykonawania funkcji
RETURN (0)                               ; rekord o polu „idnum” równym parametrowi
                                         „testidnum” nie został odnaleziony.
                                         Wartość funkcji wynosi 0.
```

Funkcja ta nie wymaga specjalnego komentarza, ponieważ używa konstrukcji z poprzedniego przykładu. Cały proces polega na przechodzeniu od rekordu, do rekordu i sprawdzaniu czy pole „idnum” jest takie samo jak parametr „testidnum”. W przykładzie 4 połączymy podprogramy z dwóch poprzednich przykładów w jeden program.

### Przykład 4:

```
MODULE                                    ; deklaracja zmiennych globalnych
    TYPE idinfo =[CARD idnum,            ; numer identyfikacyjny pracownika
                  codenum                 ; kod dostępu
                  BYTE level,            ; poziom zatrudnienia
                  name]                  ; pierwsza litera imienia
    BYTE ARRAY idarray (1000)           ; obszar pamięci wystarczający na
                                         przechowanie 40 rekordów
    DEFINE recordsize = „25”,
           nameoffset = „5”
    CARD reccount =[0]
PROC fillinfo()
;++++ procedura ta wprowadza do tablicy rekordy z informacją
; o pracowniku
idinfo POINTER newrecord
BYTE POINTER nameptr
BYTE continue
    DO
        newrecord = idarray + ( reccount * recordsize)
        Print(„Numer identyfikacyjny? „)
        newrecord.idnum = InputC()
        Print(„Poziom zatrudnienia /A-Z/? „)
        newrecord.level = GetD(7)
        Print(„Kod dostępu? „)
        newrecord.codenum = InputC()
        nameptr = newrecord + nameoffset
```

```

Print („Imię i nazwisko pracownika? „)
InputS(nameptr)
reccount == +1
PutE()
Print („Wprowadzasz następny rekord /T lub N/? „)
continue = GetD(7)
PutE()
UNTIL continue = „N OR continue = „n
OD
RETURN

CARD FUNC findmatch (CARD testidnum)
    idinfo POINTER seeker
    BYTE ctr
    FOR ctr = 0 TO (reccount-1)
        DO
            seeker = idarray + (ctr * recordsize)
            IF seeker.idnum = testidnum THEN
                RETURN (seeker)
            FI
        OD
    RETURN (0)

PROC main()
;++++ procedura ta steruje procesem wpisywania i wyszukiwania danych
    idinfo POINTER recptr                ; wskaźnik do rekordu
    BYTE POINTER nameptr                ; wskaźnik do pola name
    CARD id_num,                        ; wprowadzony numer identyfikacyjny
        code_num,                      ; wprowadzony numer kodu
        keyid = [655353]                ; numer identyfikacyjny pozwalający opuścić pętlę
                                        DO-OD
    BYTE mode                            ; tryb działania

PrintE („Start...“)
PrintE („Wybierz tryb działania.“)
PrintE („X = rozszerzanie listy pracowników.“)
PrintE („A = wyszukiwanie danych o pracownikach.“)
Print („--? „)
mode = InputB()                        ; wprowadzenie trybu pracy
IF mode = „X OR mode = „x THEN ; tryb „X“
    Fillinfo()
ELSE                                     ; tryb „A“
    DO                                    ; początek pętli
        Print („Numer identyfikacyjny pracownika --? „)
        id_num = InputC()
        IF id_num = keyid THEN           ; warunek pozwalający zakończyć pętlę
            EXIT
        ELSE                               ; poprawny numer identyfikacyjny
            recptr = findmatch(id_num)    ; przeszukiwanie tablicy
            IF recptr = 0 THEN           ; nie ma takiego pracownika
                PrintE („NIE ZNALEZIONO“)
            ELSE                           ; podany numer identyfikacyjny/istnieje
                Print („Kod dostępu --? „)
                code_num = InputC( )
                IF recptr.codenum = code_num THEN ; zgodność kodów
                    nameptr = recptr + nameoffset
                    Print („Numer identyfikacyjny: „)
                    PrintCE(recptr.idnum)
                    Print („Poziom: „)
                    Put(recptr.level)
                    Print („Imię i nazwisko: „)

```

```
        PrintE (nameptr)
        PutE ()
    ELSE                                     ; niezgodność kodów
        PrintE („BŁĘDNY KOD”)
    FI                                     ; koniec testowania kodu dostępu
FI                                       ; koniec sprawdzania numeru identyfikac.
OD                                       ; koniec pętli
FI                                       ; koniec „IF mode = ...”
PrintE („Koniec pracy.”)
RETURN                                     ; koniec programu głównego
```

Procedura „main” sprawdza szereg warunków aby określić jakie mają zostać podjęte działania .Użyto do tego zagnieżdżonych instrukcji IF. Zauważcie, jak bardzo „wcięcia” w tekście programu poprawiają jego czytelność.



```

DO
incclr = chgclr           ; ustawia kolor podstawowy
  DO                     ; początek pętli UNTIL
  wsync = 0             ; czeka na koniec" wyświetlania linii
  clr = incclr          ; zmienia wyświetlany kolor
  incclr == +1
  UNTIL vcount = 128    ; koniec testowania ekranu
  OD                     ; koniec pętli UNTIL
OD                       ; koniec pętli FOR
cbgclr == +1           ; zmiana koloru podstawowego
OD
RETURN

```

Proponujemy Wam abyście uruchomili ten program i zobaczyli jaki jest efekt jego działania. Pozwoli to Wam łatwiej zrozumieć powyższy tekst.

### 9.3 Podprogramy adresujące

W paragrafie poprzednim mówiliśmy o użyciu rejestrów sprzętowych komputera Atari poprzez adresowaniu zmiennych *ACTION!* na te rejestry. Ponieważ istnieje także możliwość zdefiniowania adresu podprogramu, można w ten sam sposób odwoływać się bezpośrednio do podprogramów sprzętowych lub systemu operacyjnego i sterować samemu operacjami wejścia/wyjścia. Metody te będą omawiane w następnym paragrafie, ponieważ odnoszą się do wszystkich podprogramów w języku maszynowym niezależnie czy napisanych przez użytkownika, czy należących do systemu operacyjnego, czy też znajdujących się w pamięci ROM.

### 9.4 Asembler ACTION!

*ACTION!* pozwala wywołać podprogramy w języku maszynowym w bardzo prosty sposób. Muszą być spełnione tylko dwa warunki:

1. należy znać adres początkowy podprogramu
2. podprogram musi być zakończony przez RTS (jeżeli chce się następnie wrócić z powrotem do *ACTION!*)

Nie są to żądania zbyt wygórowane.

Do podprogramów napisanych w języku maszynowym, podobnie jak do napisanych w języku *ACTION!*, istnieje możliwość przekazywania parametrów. Kompilator zapamiętuje parametry w następujący sposób:

adres	n-ty bajt parametrów
rejestr A	1
rejestr X	2
rejestr Y	3
\$A3	4
\$A4	5
. . .	. . .
\$AF	16

Przykłady:

```

PROC CIO = $E456 (BYTE areg,xreg)
;++++ deklaruje procedurę systemową CIO. „xreg” będzie zawierał numer
; iocb pomnożony przez 16, natomiast parametr „areg” służy tylko jako
; wypełnienie aby liczba ta nie weszła do rejestru X.
PROC readchannel2()
;++++ procedura ta otwiera kanał 2 do danej nazwy pliku i wywołuje
; podprogram maszynowy CIO do odczytu tego pliku
  DEFINE buflen = „$2000”           ; długość tablicy bufora
  BYTE ARRAY filename(30),           ; tablica nazwy pliku
          buffer(buflen)             ; tablica bufora 1
  BYTE iocb2cmd = $362               ; bajt komendy iocb 2
  CARD iocb2buf = $364               ; adres początkowy bufora iocb 2

```

```

        iocb2len = $368                ; długość bufora iocb 2
PutE()
Print („Nazwa pliku --? „)
InputS(filename)                    ; pobiera nazwę pliku
Open(2, filename, 4, 0)              ; otwiera kanał 2 do odczytu
iocb2cmd = 7                         ; komenda „pobierz rekord binarny”
iocb2buf = buffer                    ; ustawia bufor iocb na bufor zadany przez użytkownika
iocb2len = buflen                     ; ustawia długość bufora iocb
CIO(0, $20)                          ; ++++ wywołanie podprogramu CIO ++++
Close(2)                              ; zamknięcie kanału 2
RETURN

```

Widzicie jakie to proste? Jest to duże ułatwienie dla tych, którzy często korzystają z podprogramów w języku asemblerowym. W ten sposób można ich używać bezpośrednio z języka wysokiego poziomu, gdzie pisanie programów jest dużo prostsze.

Prawdopodobnie nie wszyscy zrozumieli ten przykład. IOCB oznacza „Blok Sterowania Operacjami Wejścia Wyjścia”. Trochę informacji na ten temat znajduje się w części VI podręcznika, a znacznie więcej w podręczniku poświęconym systemowi operacyjnemu.

## 9.5 Dodatkowe rady odnośnie korzystania z parametrów

Parametry i sposób ich użycia omawiane były w §6.4. Wspomnieliśmy tam, że poprzez parametry nie można wyprowadzić wartości na zewnątrz podprogramu. Nie była to cała prawda. Istnieje taka możliwość jeżeli użyje się zmiennych wskaźnikowych.

Zamiast przekazywania jako parametr konkretnej zmiennej należy umieścić zamiast niej zmienną wskaźnikową ustawioną na jej adres. W podprogramie będzie można zmieniać wartości zmiennej używając do tego danej zmiennej wskaźnikowej i zmiany te będą widoczne na zewnątrz.

```

BYTE FUNC substr (BYTE ARRAY str, sub BYTE POINTER errptr, notfound)
;++++ funkcja ta szuka w tablicy „str” zadanego łańcucha „sub”.
; jeżeli zostanie on odnaleziony funkcja przyjmuje wartość równą
; indeksowi tego łańcucha. Jeżeli łańcuch „sub” jest dłuższy
; niż łańcuch w tablicy „str”, poprzez wskaźnik sygnalizowany jest błąd.
; jeżeli łańcuch „sub” nie zostanie odnaleziony, komunikat o tym jest
; przekazywany przez inny wskaźnik.
    BYTE ARRAY tempstr                ; służy do chwilowego przechowywania
                                        ; testowanego łańcucha
    BYTE ctrl,                        ; licznik pętli zewnętrznej
        ctr2                          ; licznik pętli wewnętrznej
    IF sub(0) > str(0) THEN            ; poszukiwany łańcuch jest dłuższy niż
        errorptr^ = 1                 ; łańcuch główny
    ELSE
        FOR ctrl = 1 TO str(0)        ; pętla sprawdzająca łańcuch
            DO
                IF sub(1) = str(ctrl) THEN ; sprawdzanie pierwszych znaków
                    tempstr(0) = sub(0) ; tempstr przyjmuje długość
                                        ; poszukiwanego łańcucha
                    FOR ctr2=1 TO sub (0) ; zapełnia tempstr
                        DO
                            temostr (ctr2) = str(ctrl2+ctrl1-1)
                        OD
                    IF SCompare (tempstr, sub)=00 THEN ; porównuje dwa łańcuchy
                        RETURN(ctrl) ; jeżeli są równe funkcja przyjmuje
                                        ; wartość indeksu tego łańcucha
                    FI ; koniec sprawdzania pierwszych znaków
            OD ; koniec pętli FOR
        FI
        notfound^= 1 ; łańcuch nie został odnaleziony
    RETURN (0) ; koniec funkcji substr

```

Do wywołania tej funkcji należy użyć następującego formatu:

```
<indeks>= substr(<łańcuch główny>,<łańcuch poszukiwany>,<errptr>, <notfindptr>)
```

gdzie:	<indeks>	jest indeksem określającym w którym miejscu łańcucha głównego rozpoczyna się łańcuch poszukiwany
	<łańcuch główny>	łańcuch tekstowy, w którym poszukujemy zadanego tekstu
	<łańcuch poszukiwany>	fragment tekstu, który chcemy odnaleźć
	<errptr>	zmienna wskaźnikowa do zmiennej określającej czy nie wystąpił błąd (tzn. gdy tekst poszukiwany jest dłuższy niż łańcuch główny).
	<notfindptr>	wskaźnik do baj tu określającego czy tekst „ został odnaleziony

W ten oto sposób można uzyskać z procedury nowe wartości i jest to często bardziej efektywne niż stosowanie zmiennych globalnych.

# Część V                      Kompilator *ACTION!*

<b>Rozdział 1</b>	<b>Wstęp</b>	<b>77</b>
1.1	Słownik	77
1.2	Dyrektywy kompilatora	77
<b>Rozdział 2</b>	<b>Sposób działania kompilatora</b>	
	<b>- przydzielanie pamięci</b>	<b>78</b>
2.1	Komentarze, SET, DEFINE	78
2.2	Przydzielanie pamięci	78
2.3	Podprogramy	78
2.4	Programy dołączane (INCLUDE)	79
2.5	Dodatkowe zmienne globalne (MODULE)	79
2.6	Tablice symboli	79
<b>Rozdział 3</b>	<b>Korzystanie z menu opcji</b>	<b>80</b>
<b>Rozdział 4</b>	<b>Uwagi techniczne</b>	<b>81</b>
4.1	Obliczenia matematyczne	81
4.2	Zgodność typów i kontrola przekroczenia zakresu tablicy	81
4.3	Zastrzeżenia odnośnie kanału 7	81
4.4	Przekroczenie objętości pamięci	81



# Rozdział 1                      Wstęp

Atari BASIC jest bardzo wygodny w użyciu ponieważ jest stosunkowo zbliżony do języka angielskiego i można natychmiast testować napisane programy bez wykonywania żadnych dodatkowych operacji. Niestety jego najpoważniejszą wadą jest to, że programy napisane w tym języku wykonują się dosyć powoli. Wiąże się to z tym, że w czasie procesu obliczeniowego każda wykonywana linia musi być sprawdzana i zamieniana na kod maszynowy. Pod tym względem *ACTION!* znacznie przewyższa BASIC. Zanim program zostanie uruchomiony musi przejść przez proces kompilacji. Podczas tego procesu kompilator analizuje program linia po linii i jeżeli jest poprawny pod względem składni języka, jest przekształcany na kod maszynowy, a zmienne globalne i lokalne przyjmują konkretne adresy pamięci. Dopiero tak przekształcony program może być uruchomiony, a jego czas wykonania będzie znacznie krótszy niż w przypadku BASIC-a.

## 1.1                      Słownik

W rozdziale tym używa się terminów, które były już omówione w części IV podręcznika. Oto ich lista:

termin	komentarz
identyfikator	dowolny poprawny pod względem budowy identyfikator
wartość	dowolna wartość heksadecymalna lub dziesiętkowa
stała kompilacji	wyznacza adres identyfikatora
adres	miejsce w pamięci komputera

## 1.2                      Dyrektywy kompilatora

Dyrektywy kompilatora były dosyć dokładnie omówione w części IV w rozdziale 7. W tej części zostaną one przypomniane oraz podamy kilka nowych informacji na ich temat. Przypominamy, że dyrektywy kompilatora są wykonywane podczas kompilacji, a nie po uruchomieniu programu. Dlatego też, nie należy ich używać do zmiany parametrów operacyjnych w czasie procesu obliczeniowego.

## Rozdział 2                      Sposób działania kompilatora – przydzielanie pamięci

W rozdziale tym omówimy w jaki sposób kompilator przydziela pamięć dla programu, jego zmiennych, podprogramów i tablic symboli.

Pierwszą rzeczą jaką wykonuje kompilator jest podjęcie decyzji gdzie zostanie umieszczony wygenerowany kod programu. Dokonuje się to przez przejrzanie pamięci począwszy od komórki 14. Wartości CARD umieszczone od tego miejsca zawierają adres początkowy wolnej pamięci. Adres ten będzie się zmieniał w zależności od rozmiaru bufora edytora (zob. dodatek B). Jeżeli nie zostanie to określone przez użytkownika w inny sposób, kompilator umieści kod wynikowy programu począwszy od tego adresu.

Można nakazać aby kompilator umieścił wygenerowany kod w konkretnym miejscu pamięci. Należy w tym celu przed procesem kompilacji na poziomie monitora systemu wykonać dwie komendy”:

```
SET 14 = <adres>
SET $491 = <adres>
```

### 2.1                      Komentarze, SET, DEFINE

Dla komentarzy, dyrektywy SET oraz dyrektywy DEFINE umieszczonych w tekście programu nie jest generowany kod maszynowy. Dzieje się tak ponieważ nie wpływają one bezpośrednio na proces obliczeniowy, a więc nie są potrzebne.

### 2.2                      Przydzielanie pamięci

Informacja o zmiennych jest zapamiętywana przez kompilator w dwóch różnych miejscach – w wygenerowanym kodzie programu oraz w tablicy symboli. Tablica ta zostanie omówiona później.

Zmienne są z reguły zapamiętane na początku kodu maszynowego. Są to zmienne zadeklarowane przed pierwszym podprogramem (zmienne globalne). Mogą być one używane we wszystkich następujących podprogramach i nie trzeba ich dodatkowo deklarować w innych miejscach. Zmiennej zostaje przypinany obszar pamięci w zależności od jej typu. Tablica poniżej powinna ułatwić Wam zrozumienie procesu przydziału pamięci:

typ danych	obszar pamięci	komentarz
BYTE	1 bajt	typ podstawowy
CHAR	1 bajt	typ podstawowy
CARD	2 bajty	typ podstawowy
INT	2 bajty	typ podstawowy
ARRAY	rozmiar typu podstawowego pomnożony przez ilość elementów tablicy	typ złożony
TYPE	suma rozmiarów typów podstawowych w danej deklaracji	typ złożony
łańcuch	wszystkie znaki łańcucha plus poprzedzający je bajt określający długość tego łańcucha	każdy łańcuch jest umieszczany oddzielnie nawet jeżeli jest wstawiany pod ten sam identyfikator

### 2.3                      Podprogramy

Kompilator przydziela pamięć dla podprogramów (procedur i funkcji) po obszarze zajęтым przez zmienne globalne. Zmienne lokalne danego podprogramu poprzedzają w wygenerowanym kodzie pozostałe jego instrukcje.

## 2.4 Programy dołączane (INCLUDE).

Mówiliśmy, że za pomocą komendy `INCLUDE` można do pisanego programu dołączać programy znajdujące się na kasie lub dyskietce. Oczywiście włączany tekst nie może pozostawać w konflikcie z tekstem aktualnie przetwarzanym (mogą to być np. niewłaściwie użyte identyfikatory). Jeżeli zostanie wykryty taki błąd, w obszarze komunikatów pojawi się odpowiedni komunikat i zostanie wysłany sygnał dźwiękowy.

## 2.5 Dodatkowe zmienne globalne (MODULE).

Dyrektywa `MODULE` pozwala deklorować zmienne globalne w innych miejscach programu, a nie tylko na jego początku. Obszar przydzielany przez kompilator tym zmiennym będzie się znajdować na końcu poprzedzającego je podprogramu. Oczywiście, identyfikatory te są również umieszczane w tablicy symboli globalnych.

## 2.6 Tablice symboli

Kompilator *ACTION!* tworzy dwie tablice symboli -- jedną dla zmiennych globalnych, drugą dla zmiennych lokalnych ostatniego kompilowanego podprogramu. Tablice symboli są dostępne z poziomu monitora poprzez komendy „?”, „+” i `SET` (zob. część III). Są one także używane przez kompilator za każdym razem, gdy jest potrzebny adres danej zmiennej.

Dla tablic symboli kompilator przydziela 8 stron pamięci (2 KB). Nie jest możliwe przejście na poziom monitora podczas wykonywania programu w celu przejrzenia zawartości tych tablic, ponieważ przy takim przejściu są one wymazywane.

## Rozdział 3 Korzystanie z menu opcji

Menu opcji pozwala zmieniać warunki działania kompilatora. Wszystkie opcje są omówione poniżej, w części III podręcznika, a także w dodatku G.

### Zwiększenie prędkości działania kompilatora:

Czas kompilacji można zmniejszyć prawie o 30% przez wyłączenie ekranu podczas operacji dyskowych wejścia/wyjścia i kompilacji programu.

Należy w tym celu w odpowiedzi na pytanie „Screen?” nacisnąć „N <RETURN>”.

**UWAGA:** Ekran będzie wyłączony również dla innych funkcji systemu, dlatego też należy po zakończonej kompilacji z powrotem go włączyć.

### Wyłączanie dźwięku:

Podczas testowania nowego programu może wystąpić dużo błędów i każdorazowo będzie to sygnalizowane sygnałem dźwiękowym. Aby wyłączyć ten sygnał należy w odpowiedzi na pytanie „Bell?” nacisnąć N <RETURN>.

### Spowodowanie, aby kompilator zwracał uwagę na rodzaj wprowadzanych liter:

Można zażądać, aby kompilator dawał znać użytkownikowi, gdy ten wprowadzi słowa kluczowe języka *ACTION!* małymi literami alfabetu (wiąże się to z poprawieniem czytelności programu). Można również zażyczyć sobie, aby identyfikatory pisane różnym rodzajem liter odpowiadały innym zmiennym. Dla obydwóch powyższych przypadków, w odpowiedzi na pytanie „Case sensitive?” należy nacisnąć „Y <RETURN>”.

### Listowanie kodu wynikowego:

Można rozkazać kompilatorowi, aby listował każdą linię programu, którą przetwarza. Może to się wydawać zbędne ponieważ większość wykrytych błędów, jest wyświetlanych na ekranie podczas procesu kompilacji. Opcja ta przydaje się jeżeli ma się długie programy które włączają podprogramy z innych źródeł (komenda INCLUDE).

Dzięki tej opcji można uzyskać w takich przypadkach kompletny kod źródłowy razem z listingiem. Istnieje możliwość, aby listowanie odbywało się na drukarce (zob. część VI §7.9). Aby opcja ta stała się aktywna, w odpowiedzi na pytanie „List?” należy nacisnąć „Y <RETURN>”.

## Rozdział 4 Uwagi techniczne

### 4.1 Obliczenia matematyczne

Założmy, że zmienna typu `BYTE` ma wartość 255 i dodajemy do niej 1.

Jej nowa wartość będzie wynosiła 0, a nie jak można by tego oczekiwać 256. Wynika to z tego, że pojedynczy bajt może tylko zawierać liczby od 0 do 255. Jest to podobne do tego, gdy w systemie dziesiętkowym mając na ekranie tylko dwie pozycje na wyświetlenie liczby dodajemy do 99 jeden. Będziemy wiedzieli, że wynik równa się 100, ale na ekranie będzie widoczne „00”. Sytuacja taka określana jest jako przepełnienie matematyczne i kompilator *ACTION!* tego nie wykrywa.

Podobnie, jeżeli od 0 odejmiemy 1 otrzymamy 255.

Jak wspomniano w części IV w §4.2 niektóre z operatorów matematycznych dają wynik w określonym typie i używając takiej automatycznej zmiany typu można czasami uniknąć wyżej opisanych sytuacji.

### 4.2 Zgodność typów i kontrola przekroczenia zakresu tablicy

Należy zachować dużą ostrożność podczas pisania programu ponieważ kompilator nie wykrywa błędu przekroczenia zakresu tablicy. Zostało to zrobione z rozważą, aby umożliwić większą elastyczność operacji na danych. Proponujemy aby pisząc programy umieszczać w nich samemu taką kontrolę i instrukcje drukujące komunikaty o wystąpieniu takiej sytuacji.

### 4.3 Zastrzeżenia odnośnie kanału 7

Po wprowadzeniu systemu *ACTION!* zostaje automatycznie otwarty kanał 7 przeznaczony do odczytu z klawiatury (K:). Kanału tego można następnie używać do tego celu, ale nie można zmieniać jego atrybutów przez ponowne otwarcie lub zamknięcie.

### 4.4 Przekroczenie objętości pamięci

Może się zdarzyć, że pracując z dużym programem nagle zostanie przekroczona pojemność dostępnej pamięci. Zależnie od momentu, w którym pojawił się ten błąd, możliwe są następujące działania:

- jeżeli zdarzyło się to podczas pracy z edytorem należy natychmiast zapamiętać swój plik (`<CTRL> <SHIFT> W`), przejść na poziom monitora i uruchomić cały system *ACTION!* od początku (`BOOT`). Następnie można wrócić pod edytor i wprowadzić ponownie swój plik do komputera.
- jeżeli było to podczas kompilacji należy przejść pod edytor, zapamiętać program, przejść na poziom monitora, uruchomić ponownie cały system i skompilować swój program wprost z urządzenia, na którym został zapamiętany (dyskietka, kaseta itd.).

# Część VI Biblioteka podprogramów ***ACTION!***

<b>Rozdział 1</b>	<b>Wstęp</b>	<b>84</b>
1.1	Słownik	84
1.2	Opisy podprogramów	84
<b>Rozdział.2</b>	<b>Podprogramy wyprowadzające dane</b>	<b>85</b>
2.1	Procedury Print	85
2.1.1	Drukowanie łańcuchów	85
	PROC Print	
	PROC PrintE	
	PROC PrintD	
	PROC PrintDE	
2.1.2	Drukowanie wartości typu BYTE	86
	PROC PrintB	
	PROC PrintBE	
	PROC PrintBD	
	PROC PrintDDE	
2.1.3	Drukowanie wartości typu CARD	86
	PROC PrintC	
	PROC PrintCS	
	PROC PrintCD	
	PROC PrintCDE	
2.1.4	Drukowanie wartości typu INT	86
	PROC PrintI	
	PROC PrintIE	
	PROC PrintID	
	PROC PrintIDE	
2.1.5	PROC Printf - formatowanie wyjścia	87
2.2	Podprogramy Put	87
	PROC Put	
	PROC PutE	
	PROC PutD	
	PROC PutDE	
<b>Rozdział 3</b>	<b>Podprogramy wprowadzania danych</b>	<b>88</b>
3.1	Wprowadzanie danych numerycznych	88
	BYTE FUNC InputB	
	CARD FUNC InputC	
	INT FUNC InputI	
	BYTE FUNC InputBD	
	CARD FUNC InputCD	
	INT FUNC InputID	
3.2	Wprowadzanie łańcuchów	88
	PROC Inputs	
	PROC InputSD	
	PROC InputMD	
3.3	CHAR FUNC GetD	89
<b>Rozdział 4</b>	<b>Podprogramy manipulujące plikami</b>	<b>90</b>
4.1	PROC Open	90
4.2	PROC Close	90
4.3	PROC XIO	90

4.4	PROC Note	91
4.5	PROC Point	91
<b>Rozdział 5 Grafika i manipulatory gier</b>		<b>92</b>
5.1	PROC Graphics	92
5.2	PROC SetColor	92
5.3	BYTE color	93
5.4	PROC Plot	93
5.5	PROC DrawTo	94
5.6	PROC Fill	94
5.7	PROC Position	94
5.8	BYTE FUNC Locate	94
5.9	PROC Sound	95
5.10	PROC SndRst	95
5.11	BYTE FUNC Paddle	95
5.12	BYTE FUNC PTrig	96
5.13	BYTE PUNC Stick	96
5.14	BYTE FUNC Strig	96
<b>Rozdział 6 Operacje na łańcuchach tekstowych</b>		<b>97</b>
6.1	Podprogramy porównywania, kopiowania i wstawiania łańcuchów	97
6.1.1	INT FUNC SCompare	97
6.1.2	PROC SCopy	97
6.1.3	PROC SCopyS	97
6.1.4	PROC SAssign	98
6.2	Zamiana liczby na łańcuch tekstowy	98
	PROC StrB	
	PROC StrC	
	PROC StrI	
6.3	Zamiana łańcucha tekstowego na liczbę	98
	BYTE FUNC ValB	
	CARD FUNC ValC	
	INT FUNC ValI	
<b>Rozdział 7 Podprogramy różne</b>		<b>99</b>
7.1	BYTE FUNC Rand	99
7.2	PROC Break	99
7.3	PROC Error	99
7.4	BYTE FUNC Peek	100
	CARD FUNC PeekC	
7.5	BYTE FUNC Poke	100
	CARD FUNC PokeC	
7.6	PROC Zero	100
7.7	PROC SetBlock	101
7.8	PROC MoveBlock	101
7.9	BYTE device	101
7.10	BYTE TRACE	101
7.11	BYTE LIST	101
7.12	BYTE ARRAY EOF (8)	102

# Rozdział 1                      Wstęp

Biblioteka *ACTION!* dostarcza znaczną ilość podprogramów graficznych i realizujących operacje wejścia/wyjścia. Dzięki temu użytkownik nie musi ich sam rozpisywać. Pozwala to zaoszczędzić dużo czasu szczególnie początkującym programistom.

## 1.1                      Słownik

Większość z terminów używanych w tej części podręcznika zostało już zdefiniowanych wcześniej i tylko dwa wymagają omówienia — IOCB oraz kanał.

IOCB oznacza „Blok Sterowania Operacjami Wejścia/Wyjścia” (*Input Output Control Block*).

CIO (Central I/O) używa bloków IOCB do wykonywania funkcji wejścia/wyjścia. Podprogramy biblioteczne *ACTION!* przeznaczone do tych funkcji ustawiają IOCB w taki sposób, aby przekazać do CIO jakie czynności mają być wykonane, a następnie zostaje wprost wywołana procedura CIO.

Bloki IOCB są ponumerowane od 0 do 7. Jeżeli używa się podprogramów, które wymagają numeru kanału, numer ten jest numerem IOCB, który zawiera informację o danym urządzeniu peryferyjnym. Nie oznacza to, że konkretny IOCB steruje konkretnym urządzeniem peryferyjnym. Musi on zostać wcześniej odpowiednio ustawiony. Dzieje się to za pomocą procedury bibliotecznej „Open”.

Jeżeli napotkacie termin „kanał specjalny” będzie to się odnosiło do IOCB ustawionego na ekran. Oznacza to, że podprogramy wejścia/wyjścia używające „kanału specjalnego” będą pobierały i wysyłały informację z i na ekran (urządzenie „E:”).

**UWAGA:** Kanał specjalny jest kanałem 0.

**UWAGA:** Więcej informacji na temat bloków IOCB znajdziecie w podręczniku poświęconym systemowi operacyjnemu.

## 1.2                      Opisy podprogramów

Podprogramy biblioteczne są przedstawione w formie, która pozwoli łatwo zrozumieć jak ich używać i je wywoływać. Każdy z paragrafów będzie opisywał pojedynczy podprogram biblioteczny i będzie miał następującą budowę:

- cel podprogramu
- format
- parametry
- komentarz



## Rozdział 2 Podprogramy wyprowadzające dane

Biblioteka *ACTION!* dostarcza szczególnie dużej grupy podprogramów do umieszczania zarówno danych numerycznych, jak i tekstowych w dowolnym kanale.

Dwa podstawowe podprogramy — `Print` i `Put` — posiadają opcje pozwalające skierować wyprowadzanie danych na określony kanał i/lub umieszczanie znaków końca linii EOL.

### 2.1 Procedury Print

Procedury, które będziemy omawiać w tym paragrafie, wszystkie zaczynają się słowem „Print”. Dokładny rodzaj procedury będzie określała litera zapisana po tym słowie. Format:

```
Print<typ danych> {D}{E} (<parametry>)
```

gdzie:	<code>Print</code>	jest podstawową częścią nazwy procedury
	<typ danych>	określa jaki typ danych będzie wyprowadzany
	B	– dane typu BYTE
	C	– dane typu CARD
	I	– dane typu INT
		– łańcuch
	D	jest używane do zdefiniowania, na jakie urządzenie (kanał) mają być wyprowadzane dane
	E	użycie tej litery sygnalizuje, że po wyprowadzeniu danych ma być umieszczony znak końca linii EOL (RETURN)
	<parametry>	wymagane przez procedurę parametry

**UWAGA:** Zarówno „D” i ”E”, są opcjonalne, natomiast typ wyprowadzanych danych jest zawsze określony.

Tablica poniżej przedstawia wszystkie możliwe wywołania podprogramów `Print`:

	<u>łańcuchy</u>	<u>BYTE</u>	<u>CARD</u>	<u>INT</u>
<b>bez opcji</b>	<code>Print</code>	<code>PrintB</code>	<code>PrintC</code>	<code>PrintI</code>
<b>z opcją E</b>	<code>PrintE</code>	<code>PrintBE</code>	<code>PrintCE</code>	<code>PrintIE</code>
<b>z opcją D</b>	<code>PrintD</code>	<code>PrintBD</code>	<code>PrintCD</code>	<code>PrintID</code>
<b>z opcjami ED</b>	<code>PrintDE</code>	<code>PrintBDE</code>	<code>PrintCDE</code>	<code>PrintIDE</code>

Zwróćcie uwagę, że procedury zgrupowaliśmy w zależności od typu wyprowadzanych przez nie danych. Podobny podział zastosowano w następnych paragrafach omawiających dokładnie ich działanie.

Istnieje jeszcze jedna procedura `Print`, o której do tej pory nie wspomnieliśmy. Jest to `PrintF` i pozwala ona wyprowadzić wartości liczbowe oraz łańcuchy tekstowe. Zostanie ona omówiona w osobnym paragrafie.

#### 2.1.1 Drukowanie łańcuchów

cel: wszystkie cztery poniższe procedury służą do wyprowadzania łańcuchów tekstowych

```
format: PROC Print (<łańcuch> )
        PROC PrintE (<łańcuch>)
        PROC PrintD (BYTE kanał, <łańcuch>)
        PROC PrintDE (BYTE kanał, <łańcuch>)
```

parametry: <łańcuch> – jest albo stałą tekstową umieszczoną w cudzysłowach albo identyfikatorem tablicy `BYTE ARRAY`, którą chce się wydrukować jako tekst  
 <kanał> – jest numerem kanału (0 – 7)

komentarz: `Print` wyprowadza łańcuch na ekran łącznie ze znakiem końca linii  
`PrintE` wyprowadza łańcuch na ekran łącznie ze znakiem końca linii  
`PrintD` wyprowadza łańcuch na wyspecyfikowany kanał bez znaku końca linii  
`PrintDE` wyprowadza łańcuch na wyspecyfikowany kanał łącznie ze znakiem końca linii

Użycie tych procedur w programie jest bardzo proste, ale należy pamiętać, że dla procedur, które wymagają kanału, kanał ten musi zostać wcześniej otwarty.

## 2.1.2 Drukowanie wartości typu BYTE

cel: cztery poniższe procedury wypisują dane typu BYTE w formacie dziesiętnym

format:	PROC PrintB (BYTE liczba)	
	PROC PrintBE (BYTE liczba)	
	PROC PrintBD (BYTE kanał, liczba)	
	PROC PrintBDE (BYTE kanał, liczba)	
parametry:	liczba	– jest wyrażeniem arytmetycznym (stała lub nazwa zmiennej jest również wyrażeniem arytmetycznym)
	kanał	– jest numerem kanału (0-7)
komentarz:	PrintB	wyprowadza na ekran bajt bez znaku końca linii
	PrintBE	wyprowadza na ekran bajt łącznie ze znakiem końca linii
	PrintBD	wyprowadza na wyspecyfikowany kanał bajt bez znaku końca linii
	PrintBDE	wyprowadza na wyspecyfikowany kanał bajt łącznie ze znakiem końca linii

## 2.1.3 Drukowanie wartości typu CARD

cel: wyprowadzają liczby typu CARD w formacie dziesiętnym

format:	PROC PrintC (CARD liczba)	
	PROC PrintCE (CARD liczba)	
	PROC PrintCD (CARD kanał, liczba)	
	PROC PrintCDE (CARD kanał, liczba )	
parametry:	liczba	– jest wyrażeniem arytmetycznym
	kanał	– jest numerem kanału
komentarz:	PrintC	wyprowadza wartość typu CARD bez znaku końca linii na ekran
	PrintCE	wyprowadza wartość typu CARD łącznie ze znakiem końca linii na ekran
	PrintCD	wyprowadza wartość typu CARD bez znaku końca linii na wyspecyfikowany kanał
	PrintCDE	wyprowadza wartość typu CARD łącznie ze znakiem końca linii na wyspecyfikowany kanał

## 2.1.4 Drukowanie wartości typu INT

cel: wyprowadzają liczby typu INT w formacie dziesiętnym\*

format:	PROC PrintI (INT liczba)	
	PROC PrintIE (INT liczba)	
	PROC PrintID (INT kanał, liczba)	
	PROC PrintIDE (INT kanał, liczba)	
parametry:	liczba	– wyrażenie arytmetyczne
	kanał	– numer kanału
komentarz:	PrintI	wyprowadza na ekran wartość typu INT bez znaku końca linii
	PrintIE	wyprowadza na ekran wartość typu INT łącznie ze znakiem końca linii
	PrintID	wyprowadza na wyspecyfikowany kanał wartość typu INT bez znaku końca linii
	PrintIDE	wyprowadza na wyspecyfikowany kanał wartość typu INT łącznie ze znakiem końca linii

## 2.1.5 PROC PrintF

Procedura PrintF pozwala wyprowadzić liczby i łańcuchy w tej samej linii przez użycie tzw. „łańcucha kontrolnego”. Łańcuch ten określa w jaki sposób mają być wyprowadzane te dane.

cel: sformatowane wyjście danych

format: PrintF („<łańcuch kontrolny>„ , <dana> |: , <dana> :|)

argumenty: <łańcuch kontrolny> – łańcuch kontrolny okłada się ze znaków sterujących formatem oraz tekstu. Tekst jest wyprowadzany bez żadnych zmian, natomiast znaki sterujące (max. 5) zawierają informację dla wyprowadzanych <danych>.

dana – jest wyrażeniem arytmetycznym, które będzie formatowane przez przypisane mu znaki sterujące łańcucha. Pierwszy znak sterujący określa jak wyprowadzać pierwszą <dana>, drugi jak wyprowadzać następną <dana> itd.

komentarz: procedura ta pozwala wyprowadzać w sposób sformatowany dane na kanał specjalny. Do łańcucha można wstawić do 5 różnych znaków sterujących wydrukiem:

<u>znak sterujący</u>	<u>typ danych</u>
%S	wyprowadza daną jako łańcuch tekstów
%I	wyprowadza daną jako INT
%U	wyprowadza daną jako CARD bez znaku
%C	wyprowadza daną jako znak
%H	wyprowadza daną w postaci heksadecymalnej bez znaku
%%	wyprowadza znak „%”
%E	wyprowadza znak końca linii EOL (RETURN)

Zwróćcie uwagę, że dwa znaki sterujące (%E i %% ) nie mają wpływu na dane i ich nie wymagają. Są one używane do zmiany formatowania strony, a nie formatowania danych.

Dozwolone jest 5 znaków sterujących, każdy dla osobnego elementu <dana>.

## 2.2 Podprogramy Put

Grupa procedur bibliotecznych „Put” jest używana do wyprowadzania pojedynczych znaków (tzn. wyprowadzania danych typu BYTE jako znaków ATASCII). Podprogramy te używają opcji takich jak procedury „Print” dlatego nie musimy ich tutaj ponownie opisywać.

cel aby wyprowadzić pojedynczy znak ATASCII, używając określonych opcji formatu

format: PROC Put (CHAR znak)  
 PROC PutE ()  
 PROC PutD (BYTE kanał, CHAR znak)  
 PROC PutDE (BYTE kanał, CHAR znak)

parametry: znak – jest wyrażeniem arytmetycznym  
 kanał – jest numerem kanału

komentarz: Put wyprowadza na kanał specjalny znak bez znaku końca linii  
 PutE wyprowadza znak końca linii EOL (<RETURN>) na kanał specjalny  
 PutD wyprowadza na wyspecyfikowany kanał znak bez znaku końca linii  
 PutDE wyprowadza na wyspecyfikowany kanał znak łącznie ze znakiem końca linii

## Rozdział 3 Podprogramy wprowadzania danych

W rozdziale tym omówimy podprogramy, które służą do wprowadzania danych do komputera. Podobnie jak dla procedur przedstawionych w poprzednich paragrafach, typ danych i miejsce skąd są pobierane są zdefiniowane przez zastosowanie opcji.

Podprogramy wprowadzające dane można podzielić na dwie grupy: „Input” i „Get”. Podprogramy Input można z kolei podzielić na dwie kategorie: te które wprowadzają dane numeryczne i te, które wprowadzają łańcuchy. Istnieje tylko jedna procedura Get (GetD) i będzie ona omówiona w ostatnim paragrafie tego rozdziału.

### 3.1 Wprowadzanie danych numerycznych

Poniższe sześć funkcji pozwala wprowadzić daną numeryczną dowolnego typu z dowolnego kanału.

cel: do wprowadzania danych numerycznych

format:        BYTE FUNC InputB()  
               BYTE FUNC InputBD(BYTE kanał)  
               CARD FUNC InputC()  
               CARD FUNC InputCD(BYTE kanał)  
               INT  FUNC InputI()  
               INT  FUNC InputID(BYTE kanał)

parametry:  kanał        – jest numerem kanału (0-7)

komentarz:  InputB        wprowadza liczbę BYTE z kanału specjalnego  
               InputBD     wprowadza liczbę BYTE z wyspecyfikowanego kanału  
               InputC       wprowadza liczbę CARD z kanału specjalnego  
               InputCD     wprowadza liczbę CARD z wyspecyfikowanego kanału  
               InputI       wprowadza liczbę INT z kanału specjalnego  
               InputID     wprowadza liczbę INT z wyspecyfikowanego kanału

### 3.2 Wprowadzanie łańcuchów

Wprowadzanie ciągu znaków odbywa się poprzez dodanie do podstawy „wejściowej” znaku „S”. Istnieją trzy procedury biblioteczne pozwalające wprowadzać łańcuchy tekstowe. Pozwalają na wprowadzenie ciągu znaków z dowolnego kanału i/lub zdefiniowanie maksymalnej długości ciągu wejściowego.

cel: do wprowadzania danych tekstowych

format:        PROC Inputs (<łańcuch>)  
               PROC InputSD(BYTE kanał, <łańcuch>)  
               PROC InputMD(BYTE kanał, <łańcuch>, BYTE max)

parametry:  <łańcuch>    – jest identyfikatorem tablicy BYTE ARRAY  
               kanał        – numer kanału  
               max           – maksymalna długość wprowadzanego łańcucha. Jeżeli jest on dłuższy, będzie skrócony do tej długości

komentarz:  Inputs        wprowadza . łańcuch tekstowy o długości do 255 znaków z kanału specjalnego  
               InputSD     wprowadza łańcuch tekstowy o długości do 255 znaków z wyspecyfikowanego kanału  
               InputMD     wprowadza łańcuch tekstowy o długości nieprzekraczającej zadanego „max” z wyspecyfikowanego kanału

### 3.3 CHAR FUNC GetD

cel: wprowadza pojedynczy znak z danego kanału

format: CHAR PUNC GetD(BYTE kanał)

komentarz: funkcja ta jest używana do pobrania jednego znaku z urządzenia określonego przez kanał. Funkcja ta daje w wyniku wartość kodu ATASCII danego znaku.

## Rozdział 4 Podprogramy manipulujące plikami

Rozdział ten jest poświęcony tym podprogramom, które mają do czynienia z urządzeniami zewnętrznymi (drukarka, stacja dysków, pamięć kasetowa itd.). Za pomocą tych podprogramów można otwierać kanał, zamykać kanał oraz wykonywać bardziej złożone operacje na plikach dyskowych.

### 4.1 PROC Open

cel: ustawia kanał IOCB w taki sposób, że można wykonywać operacje wejścia/wyjścia na danym urządzeniu peryferyjnym.

format: PROC Open (BYTE kanał, <plik>, BYTE tryb, aux2)

parametry: kanał – jest numerem kanału (0-7)  
 plik – jest stałą tekstową (lub identyfikatorem tablicy zawierającej dany łańcuch) określającą urządzenie (D:, P:, S:, itd.), które zostanie otwarte na dany kanał (IOCB). „D:” wymaga również podania nazwy pliku.  
 Tryb – jest liczbą określającą typ operacji wejścia/wyjścia:  
     4 – tylko odczyt  
     6 – odczyt katalogu  
     8 – tylko zapis  
     9 – dopisanie danych  
     12 – odczyt/zapis  
 aux2 – wartość zależna od urządzenia (zwykle 0)

komentarz: procedura ta otwiera dany kanał dla urządzenia określonego przez plik. Istnieje możliwość ustawienia typu operacji wejścia/wyjścia. Parametrem zależnym od urządzenia jest „aux2”.

**UWAGA:** Nie należy otwierać kanału 7 ponieważ jest on używany przez system *ACTION!* do swoich własnych operacji na ekranie. Można tego kanału użyć w programie do pobrania znaków z K:, ale należy pamiętać, że jest on już otwarty.

### 4.2 PROC Close

cel: zamyka kanał IOCB

format: PROC Close (BYTE kanał)

parametry: kanał – numer kanału

komentarz: procedura ta zamyka wyspecyfikowany kanał. Powinno się zawsze na końcu programu zamykać kanały, z których się w nim korzystało.

**UWAGA:** Nie powinno się zamykać kanału 7.

### 4.3 PROC XIO

cel: zapewnia dostęp do poleceń specyficznych dla urządzenia. Na przykład, aby zmienić szybkość transmisji na urządzeniu RS232, należy w tym celu wywołać procedurę obsługi XIO dla urządzenia R:.

format: PROC XIO (BYTE kanał, 0, cmd, aux1, aux2, <plik>)

parametry: kanał – jest numerem kanału  
 cmd – odpowiada bajtowi COMMAND bloku IOCB (ICCOM dla OS/A+ i DOS XL)  
 aux1 – pierwszy bajt pomocniczy w IOCB (ICAUX1 dla OS/A+ i DOS XL)  
 aux2 – drugi bajt pomocniczy w IOCB (ICAUX2 dla OS/A+ i DOS XL)  
 plik – jest łańcuchem tekstowym określającym urządzenie standardowe (w przypadku „D:” należy także podać nazwę pliku)

komentarz: procedura ta jest przeznaczona do uzyskania dostępu do systemu operacyjnego DOS. Ci z Was, którzy znają ATARI BASIC, BASIC A+ lub BASIC XL rozpoznają, że jest to przekształcona postać instrukcji XIO.

Nie będziemy tutaj dokładnie omawiać wszystkich możliwości procedury XIO i odsyłamy Was do rozdziału 8 podręczników OS/A+ i DOS XL. Procedura XIO *ACTION!* może wykonywać wszystkie umieszczone tam komendy systemowe z wyjątkiem NOTE, POINT i operacji przesyłania różnych danych.

**UWAGA:** „0” jako drugi parametr jest obowiązkowe.

## 4.4 PROC Note

cel: wynikiem tej procedury jest aktualny sektor pliku na wyspecyfikowanym napędzie dyskowym oraz odstęp liczony w bajtach od Początku tego sektora

format: PROC Note (BYTE kanał, CARD POINTER sektor, BYTE POINTER offset)

parametry: kanał – numer kanału (0-7)  
 sektor – jest wskaźnikiem do zmiennej określającej numer sektora  
 offset – jest wskaźnikiem do zmiennej określającej przesunięcie liczone w bajtach od początku sektora

komentarz: dzięki tej procedurze uzyskuje się numer sektora dyskowego i przesunięcie (w bajtach) do następnego bajtu, który ma być odczytany lub zapisany (tzn. otrzymuje się wartość wskaźnika pliku dyskowego).

## 4.5 PROC Point

cel: pozwala ustawić wskaźnik pliku dyskowego (tzn. sektor i przesunięcie) i uzyskuje się w ten sposób swobodny dostęp do pliku.

format: PROC Point (BYTE kanał, CARD sektor, BYTE offset)

parametry: kanał – jest poprawnym numerem kanału (0-7)  
 sektor – jest poprawnym numerem sektora /1-720  
 offset – jest przesunięciem liczonym w bajtach wewnątrz tego sektora

komentarz: procedura ta pozwala ustawić wskaźnik pliku dyskowego na dowolną pozycję wewnątrz tego pliku, co pozwala z kolei na swobodny dostęp do każdej informacji w nim zapisanej

**UWAGA:** Plik dyskowy musi zostać wcześniej otwarty z trybem 12.

## Rozdział 5 Grafika i manipulatory gier

Biblioteka *ACTION!* zawiera dużą grupę podprogramów ułatwiających pisanie gier (wykorzystujących efekty graficzne i dźwiękowe), następujących paragrafach omówimy kolejno te podprogramy.

### 5.1 PROC Graphics

cel: służy do wyboru trybu graficznego

format: PROC Graphics (BYTE tryb)

parametry: tryb – jest numerem trybu graficznego

komentarz: procedura ta jest odpowiednikiem komendy o tej samej nazwie w języku BASIC i pozwala na dostęp do trybów graficznych dostępnych w komputerze ATARI.

Poniżej przedstawiamy 9 podstawowych trybów graficznych (z oknem tekstowym):

tryb graficzny	typ trybu	ilość kolumn	ilość wierszy	ilość wierszy całego ekranu	liczba kolorów
0	tekstowy	40	–	24	2
1	tekstowy	20	20	24	5
2	tekstowy	20	10	12	5
3	graficzny	40	20	24	4
4	graficzny	80	40	48	2
5	graficzny	80	40	48	4
6	graficzny	160	80	96	2
7	graficzny	160	80	96	4
8	graficzny	320	160	192	1/2

Możliwe są jeszcze tryby graficzne pochodne od przedstawionych w tabeli. Są to tryby o numerach:

n+16 – jak n, ale bez okienka tekstowego

n+32 – jak n, ale bez kasowania ekranu

n+16+32 – kombinacja n+16 i n+32, gdzie: n określa tryb przedstawiony w tabeli

### 5.2 PROC SetColor

cel: służy do zmiany koloru i poziomu jasności zapisanych w określonym rejestrze koloru

format: PROC SetColor (BYTE rejestr, kolor, jasność)

parametry: rejestr – jest jednym z pięciu rejestrów kolorów (0 - 4)

kolor – kod koloru

jasność – jest poziomem jasności

komentarz: procedura ta pozwala ustawić kolor w określonym rejestrze koloru i w ten sposób manipulować kolorami wyświetlanymi w danym trybie.

Tablica poniżej podaje możliwe do uzyskania kolory wraz z ich kodami:

<u>Kolor</u>	<u>Kod</u>
Szary	0
Złoty	1
Pomarańczowy	2
Czerwono-pomarańczowy	3
Różowy	4
Purpurowy	5
Purpurowo-niebieski	6
Niebieski	7

<u>Kolor</u>	<u>Kod</u>
Niebieski	8
Jasnoniebieski	9
Turkusowy	10
Zielono-niebieski	11
Zielony	12
Żółto-zielony	13
Pomarańczowo-zielony	14
jasnopomarańczowy	15



Tablica poniżej pokazuje, które kolory są ustawione w rejestrach jeżeli się ich nie zmieni procedurą SetColor:

<u>Rejestr</u>	<u>Kolor</u>	<u>Jasność</u>	<u>Kolor</u>
0	2	8	pomarańczowy
1	12	10	zielony
2	9	4	ciemnoniebieski
3	4	6	różowy lub czerwony
4	0	0	czarny

**UWAGA:** Jakość kolorów w, wysokim stopniu zależy od stosowanego monitora.

Jasność może być określona przez liczbę z przedziału od 0 – 15, gdzie 0 jest najciemniejszym odcieniem, a 15 najjaśniejszym.

### 5.3 BYTE color

„color” nie jest podprogramem bibliotecznym, lecz zmienną zdefiniowaną w bibliotece, której używają procedury „Plot”, „DrawTo”, „Fill”. Po ustawieniu trybu graficznego i rejestrów kolorów (używając do tego procedur „Graphics” i „SetColor”) przed rozpoczęciem rysowania należy wybrać konkretny kolor. Dokonuje się tego instrukcją przypisania:

```
color = <numer>
```

gdzie <numer> odpowiada rejestrowi kolorów zawierającemu wybrany przez nas kolor.

Tablica poniżej podaje dokładnie (w zależności od ustawionego trybu graficznego), któremu rejestrowi kolorów odpowiada jaka wartość zmiennej „color” i na co ma wpływ.

<u>Tryb graficzny</u>	<u>Rejestr kolorów</u>	<u>Zmienna „color”</u>	<u>Komentarz</u>
0 i wszystkie tryby z oknami tekstowymi	0	—	—
	1	—	—
	2	—	jasność znaku
	3	—	tło
	4	—	krawędź
1, 2	0	—	znak
	1	—	znak
	2	—	znak
	3	—	znak
	4	—	tło, krawędź
3, 5, 7	0	1	punkt graficzny
	1	2	punkt graficzny
	2	3	punkt graficzny
	3	—	—
	4	0	punkt graficzny, tło, krawędź
4, 6	0	1	punkt graficzny
	1	—	—
	2	—	—
	3	—	—
	4	0	punkt graficzny, tło, krawędź
8	0	—	—
	1	1	jasność punktu graficznego
	2	0	punkt graficzny, tło
	3	—	—
	4	—	krawędź

### 5.4 PROC Plot

cel: umieszcza kursor w podanym miejscu i wyświetla ustawiony przez zmienną biblioteczną „color” kolor.

format: PROC Plot(CARD kolumna, BYTE wiersz)

parametry: kolumna – współrzędna x rysowanego punktu  
wiersz – współrzędna y

komentarz: procedura ta jest używana w trybach graficznych 3-8 do umieszczania na ekranie punktu o współrzędnych x,y. Wielkość wyświetlanego punktu zależy od trybu graficznego, a kolor tego punktu od aktualnej wartości zmiennej „color”.

## 5.5 PROC DrawTo

cel: służy do narysowania linii pomiędzy punktem określonym poprzednią instrukcją Plot, aadaną pozycją.

format: PROC DrawTo(CARD kolumna, BYTE wiersz)

parametry: kolumna – współrzędna x końca rysowanej linii  
wiersz – współrzędna y końca rysowanej linii

komentarz: procedura ta jest używana w trybach graficznych 3-8 do rysowania linii od punktu o współrzędnych określonych ostatnią instrukcją Plot do punktu określonego przez parametry. Kolor linii zależy od aktualnej wartości zmiennej „color”.

## 5.6 PROC Fill

cel: zapełnia określonym kolorem obszar zdefiniowany przez parametry ostatniej procedury Plot i parametry procedury Fill

format: PROC Fill(CARD kolumna, BYTE wiersz)

parametry: kolumna – jest współrzędną x z prawego dolnego rogu zapełnianego obszaru  
wiersz – jest współrzędną y z prawego dolnego rogu zapełnianego obszaru

komentarz: procedura ta pozwala w trybach graficznych 5-8 tworzyć na ekranie jednokolorowe prostokąty w miejscu zdefiniowanym przez parametry bezpośrednio wcześniej użytej instrukcji Plot i współrzędne prawego dolnego rogu prostokąta podane w procedurze Fill. Kolor zależy od aktualnej wartości zmiennej „color”.

## 5.7 PROC Position

cel: umieszcza kursor w podanym miejscu ekranu

format: PROC Position(CARD kolumna, BYTE wiersz )

parametry: kolumna – współrzędna x  
wiersz – współrzędna y

komentarz: procedura ta w dowolnym trybie graficznym umieszcza na ekranie kursor w miejscu określonym przez parametry. Podprogramy biblioteczne Print, Put, Input oraz Get korzystają następnie z rejestru kursora ustawionego tą procedurą.

## 5.8 BYTE FUNC Locato

cel: określa kolor lub znak w danym miejscu ekranu

format: PROC BYTE FUNC Locato(CARD kolumna, BYTE wiersz )

parametry: kolumna – jest poprawnym w danym trybie graficznym numerem kolumny  
wiersz – jest poprawnym w danym trybie graficznym numerem wiersza

komentarz: podprogram ten daje w wyniku kod ATASCII znaku lub numer koloru znajdującego się w podanym miejscu ekranu. Rejestry tego podprogramu są zwiększane w taki sposób, aby wskazywały sąsiednią (w poziomie) pozycję, ekranu (jeżeli była testowana ostatnia pozycja w linii, rejestry te będą wskazywać pierwszą pozycję w linii następnej). Z rejestrów tych korzystają również podprogramy Get, Put, Print i Input.

## 5.9 PROC Sound

cel: udostępnia możliwości dźwiękowe komputera ATARI

format: PROC Sound (BYTE generator, częstotliwość, brzmienie, głośność)

parametry: generator – jest jednym z czterech dostępnych na Atari generatorów dźwięku (0-3)  
 częstotliwość – jest okresem generowanego dźwięku. Niższa wartość daje wyższy ton.  
 brzmienie – dźwięk, w Atari może mieć różne brzmienie. Dla wartości 10 lub 14 otrzymujemy czyste tony. Dla pozostałych wartości z zakresu 0-14 otrzymuje się dźwięki mniej lub bardziej zniekształcone.  
 głośność – jest poziomem dźwięku (0-16)

komentarz: procedura ta, podobnie jak instrukcja o tej samej nazwie w języku BASIC, pozwala sterować aparaturą generującą dźwięk w komputerze Atari. Do tworzenia muzyki parametr „brzmienie” powinien być ustawiony na 10. Tablica poniżej pokazuje jakie nuty dają poszczególne wartości parametru „częstotliwość”:

	<u>okres</u>	<u>nuta</u>		<u>okres</u>	<u>nuta</u>
<b>Dźwięki wysokie</b>	29	C	91	F	
	31	H	96	E	
	33	A# lub B $\flat$	102	D# lub E $\flat$	
	35	A	108	D	
	37	G# lub A $\flat$	114	C# lub D $\flat$	
	40	G	<b>Środkowe C</b>		
	42	F# lub G $\flat$	121	C	
	45	F	128	H	
	47	E	136	A# lub H $\flat$	
	50	D# lub E $\flat$	144	A	
	53	D	153	G# lub A $\flat$	
	57	C# lub D $\flat$	162	G	
	60	C	173	F# lub B $\flat$	
	64	H	182	F	
	68	A# lub H $\flat$	<b>Dźwięki niskie</b>		
72	A	193	E		
76	G# lub A $\flat$	204	D# lub B $\flat$		
81	G	217	D		
85	F# lub G $\flat$	230	C# lub B $\flat$		
		243	C		

## 5.10 SndRst

cel: wyłącza generatory dźwięku

format: PROC SncLRst ()

parametry: brak

komentarz: powoduje, że wszystkie generatory dźwięku przestają, wysyłać dźwięk

## 5.11 BYTE FUNC Paddle

cel: służy do określenia aktualnej wartości (pozycji) jednego z ”wiosłek”. Ze względu na brak jednoznacznej polskiej nazwy będziemy w dalszym tekście używać nazwy angielskiej – *paddle*.

format: BYTE FUNC Paddle (BYTE port)

parametry: `port` – jest numerem portu, do którego jest podłączony

komentarz: w wyniku działania tej funkcji otrzymuje się aktualną wartość wyspecyfikowanego portu manipulatora paddle

## 5.12 BYTE FUNC PTrig

cel: służy do określenia czy klawisz FIRE manipulatora paddle jest wciśnięty

format: `BYTE FUNC PTrig (BYTE port)`

parametry: `port` – jest numerem portu, do którego jest podłączony manipulator paddle

komentarz: funkcja przyjmuje wartość 0, jeżeli klawisz FIRE jest wciśnięty, w przeciwnym przypadku przyjmuje wartość różną od zera.

## 5.13 BYTE FUNC Stick

cel: daje aktualną wartość (pozycję) manipulatora joystick.

format: `BYTE FUNC Stick (BYTE port)`

parametry: `port` – jest numerem portu (0-3) joysticka, którego pozycja jest testowana

komentarz: jako wynik otrzymujemy liczbę zależną od pozycji joysticka tak jak pokazuje poniższy rysunek:

## 5.14 BYTE FUNC STrig

cel: daje informację czy klawisz FIRE joysticka jest wciśnięty

format: `BYTE FUNC STrig (BYTE port)`

parametry: `port` – jest numerem portu (0-3), do którego jest podłączony testowany joystick

komentarz: funkcja przyjmuje wartość 0, jeżeli klawisz FIRE joysticka jest naciśnięty, w przeciwnym przypadku funkcja przyjmuje wartość różną od zera.

## Rozdział 6 Operacje na łańcuchach tekstowych

### 6.1 Podprogramy porównywania, kopiowania i wstawiania łańcuchów

Podprogramy zamieszczone w następujących czterech paragrafach ułatwiają operacje na zmiennych łańcuchowych. Należy tylko pamiętać, że maksymalna długość łańcucha tekstowego wynosi 255 znaków.

#### 6.1.1 INT FUNC SCompare

cel: porównanie alfabetyczne dwóch łańcuchów tekstowych.

format: INT FUNC SCompare (<łańcuch 1>, <łańcuch 2>)

parametry: <łańcuch 1> – jest tekstem w cudzysłowach lub identyfikatorem tablicy CHAR ARRAY zawierającej łańcuch  
<łańcuch 2> – j.w.

komentarz: funkcja daje w wyniku wartość zależną od poniższej tablicy:

<u>wynik porównania</u>	<u>wartość funkcji</u>
<łańcuch 1> < <łańcuch 2>	ujemna
<łańcuch 1> = <łańcuch 2>	0
<łańcuch 1> > <łańcuch 2>	dodatnia

Porównywanie jest alfabetyczne, tak więc funkcja ta może być wykorzystywana w sortowaniu alfabetycznym łańcuchów tekstowych.

#### 6.1.2 PROC SCopy

cel: kopiuje jeden łańcuch tekstowy do drugiego

format: PROC SCopy(<łańcuch docelowy>, <łańcuch źródłowy>)

parametry: <łańcuch docelowy> – jest identyfikatorem tablicy CHAR ARRAY do której będzie kopiowany łańcuch  
<łańcuch źródłowy> – jest tekstem w cudzysłowach lub identyfikatorem tablicy CHAR ARRAY zawierającej łańcuch, który będzie kopiowany

komentarz: procedura ta kopiuje zawartość łańcucha źródłowego do określonego łańcucha docelowego. Jeżeli jest on krótszy niż łańcuch źródłowy, tekst zostanie obcięty.

**RADA:** Nie definiuj wymiaru łańcucha docelowego i unikniesz w ten sposób takich sytuacji.

#### 6.1.3 PROC SCopyS

cel: kopiuje część jednego łańcucha tekstowego do drugiego

format: PROC SCopyS(<łańcuch doc.>, <łańcuch źródł.>, BYTE start, stop)

parametry: <łańcuch doc.> – identyfikator łańcucha (CHAR ARRAY), do którego będzie kopiowany tekst  
<łańcuch źródł.> – jest tekstem w cudzysłowach lub identyfikatorem tablicy CHAR ARRAY użytej jako kopiowany łańcuch tekstowy  
start – miejsce w łańcuchu źródłowym, od którego będzie kopiowany tekst  
stop – miejsce w łańcuchu źródłowym, do którego będzie kopiowany tekst. Jeżeli „stop” jest większy niż długość całego łańcucha, przyjmuje się, że jest on równy tej długości.

komentarz: procedura ta kopiuje zawartość łańcucha źródłowego, począwszy od elementu określonego przez „start” do elementu określonego przez „stop”, do łańcucha docelowego. Jak widać jest to procedura podobna do SCOPY, z tym, że kopiowana jest tylko część łańcucha źródłowego, a nie całość.

### 6.1.4 PROC SAssign

cel: kopiuje jeden łańcuch tekstowy do części drugiego

format: PROC SAssign (<łańcuch doc.>, <łańcuch źródł.>, BYTE start, stop)

parametry: <łańcuch doc.> - identyfikator łańcucha (CHAR ARRAY), do której będzie kopiowany tekst  
 <łańcuch źródł.> - tekst w cudzysłowach lub identyfikator tablicy CHAR ARRAY użytej jako kopiowany łańcuch tekstowy  
 start - miejsce w łańcuchu docelowym, od którego będzie kopiowany tekst  
 stop - miejsce w łańcuchu docelowym, do którego będzie kopiowany tekst. Jeżeli „stop” jest większy niż długość całego łańcucha, przyjmuje się, że jest on równy tej długości.

komentarz: procedura ta jest używana do kopiowania jednego łańcucha (źródłowego) do części innego (docelowego). Kopiowany tekst zostanie umieszczony w miejscu określonym przez parametry „start” i „stop”. Jeżeli obszar w łańcuchu docelowym przeznaczony na kopiowany tekst (stop-start+1) jest większy niż długość całego łańcucha źródłowego, parametr „stop” przyjmie taką wartość, że obszar ten będzie równy tej długości.

## 6.2 Zamiana liczby na łańcuch tekstowy

Poniższe trzy procedury dokonują konwersji liczby podanej przez parametr na łańcuch tekstowy. Każda z nich jest przeznaczona dla innego typu liczby.

cel: zamiana liczby na łańcuch tekstowy

format: PROC StrB (BYTE liczba, <łańcuch>)  
 PROC StrC (CARD liczba, <łańcuch>)  
 PROC StrI (INT liczba, <łańcuch>)

parametry: liczba - jest wyrażeniem arytmetycznym (może to być również stała lub nazwa zmiennej)  
 łańcuch - identyfikator tablicy CHAR ARRAY

komentarz: procedury te dają w wyniku wartości BYTE, CARD lub INT w łańcuchu tekstowym stworzonym z cyfr danej liczby

## 6.3 Zamiana łańcucha tekstowego na liczbę

cel: zamiana łańcucha stworzonego z cyfr na liczbę

format: BYTE FUNC ValB(<łańcuch>)  
 CARD FUNC ValC(<łańcuch>)  
 INT FUNC ValI(<łańcuch>)

parametry: łańcuch - jest tekstem w cudzysłowach lub identyfikatorem tablicy CHAR ARRAY zawierającej łańcuch składający się z samych cyfr (0-9)

komentarz: funkcje te przyjmują wartość liczbową (BYTE, CARD lub INT zależnie od użytej funkcji) zawartą w danym łańcuchu tekstowym.

## Rozdział 7 Podprogramy różne

W rozdziale tym zostaną opisane następujące podprogramy:

Rand	– generator liczb losowych
Break	– zatrzymanie wykonywania programu
Error	– procedura systemowa „obsługująca” błędy
Peek	– odczyt zadanej komórki pamięci
Poke	– zapis zadanej komórki pamięci
Zero	– zeruje zadany obszar pamięci
SetBlock	– zapełnia wyspecyfikowany obszar pamięci daną wartością
MoveBlock	– przesuwa bloki pamięci
Device	– zmienna „urządzenie specjalne” (default)
Trace	– steruje opcją kompilacji TRACE
List	– steruje opcją kompilacji LIST
EOF	– zawiera status EOF wszystkich kanałów

### 7.1 BYTE FUNC Rand

cel: generuje liczbę losową

format: BYTE FUNC Rand (BYTE zakres)

parametry: zakres – jest górnym ograniczeniem generowanej liczby

komentarz: funkcja ta daje w wyniku liczbę losową z przedziału od 0 do („zakres”-1). Jeżeli w podparametr „zakres” zostanie wstawione 0, przyjmowany jest przedział 0-255.

### 7.2 PROC Break

cel: zatrzymuje wykonywanie programu

format: PROC Break()

parametry: brak

komentarz: procedura ta pozwala zatrzymać wykonywanie programu i dzięki temu umożliwia jego tekstowanie (można np. sprawdzić wartości niektórych zmiennych). Po wykonaniu komendy monitora „PROCEED” program będzie kontynuowany od następnej instrukcji,

### 7.3 PROC Error

Jest to procedura, którą wywołuje sam system *ACTION!* (lub CIO) po wykryciu błędu. Można spowodować aby *ACTION!* po wykryciu błędu wykonał czynności określone przez użytkownika. W tym celu należy do tekstu swojego programu włączyć następujące instrukcje:

```
PROC MyError (BYTE errcode)
;++++ jest to procedura użytkownika obsługi błędów.
; Parametr <errcode> zawiera kod błędu
; przekazywany przez system ACTION!
;
; tekst własnej procedury obsługi błędów
;
RETURN      ;koniec procedury MyError

PROC main()                ; program główny użytkownika
    CARD temperr           ; przechowuje adres systemowej procedury obsługi błędów
                           (PROC Error)
    temperr = Error        ; zapisanie w zmiennej temperr adresu procedury systemowej
                           PROC Error
    Error = MyError        ; spowodowanie, że adres procedury systemowej Error będzie
```

wskazywał na procedurę MyError napisaną przez użytkownika

```

;
; pozostały tekst programu głównego użytkownika
;
Error = temperr           ; spowoduje, że po wykryciu błędów ACTION! Będzie ponownie
                          ; wykonywał własną procedurę systemową, a nie napisaną przez
                          ; użytkownika
RETURN                   ; koniec programu

```

Tak więc, wszystko co należy zrobić, to napisanie własnej procedury (o dowolnej nazwie) obsługującej wykryte przez system błędy i zamiana wskaźnika do procedury systemowej tak, aby wskazywał Twoją procedurę. Procedury tej nie trzeba wywoływać samemu w tekście programu, ponieważ będzie to robił system ACTION! każdorazowo po wykryciu błędu wykonania programu.

Zauważcie, że po zakończonych obliczeniach powinno ustawić się ten wskaźnik z powrotem na procedurę systemową PROC Error, ponieważ w dalszej pracy system ciągle odwoływałby się do procedury napisanej przez użytkownika.

**UWAGA:** Wstawianie własnej procedury obsługi błędów powinno być robione z dużą ostrożnością ponieważ pominięcie w niej czegoś może doprowadzić do załamania systemu.

## 7.4 BYTE FUNC Peek i CARD FUNC PeekC

cel: przyjmuje wartość (BYTE lub CARD) danego miejsca pamięci

format: BYTE FUNC Peek(CARD adres )  
CARD FUNC PeekC(CARD adres)

parametry: adres – określa adres w pamięci komputera, z którego chce się odczytać wartość

komentarz: funkcje te pozwalają „obejrzeć” zawartość pamięci w czasie wykonywania programu i dają w wyniku odpowiednio wartości BYTE lub CARD.

## 7.5 PROC Poke i PROC PokeC

cel: służy do umieszczenia w danym miejscu pamięci nowej wartości /BYTE lub CARD/

format: PROC Poke (CARD adres, BYTE wartość)  
PROC PokeC (CARD adres, wartość )

parametry: adres – miejsce w pamięci, do którego ma być wstawiona nowa wartość  
wartość – wartość, którą się chce umieścić w miejscu określonym przez parametr „adres”. Procedura PokeC zapamiętuje wartość typu CARD w komórkach określonych przez „adres” i „adres +1” (w kolejności LSB, MSB).

komentarz: procedury te pozwalają zmienić zawartość pamięci podczas wykonywania programu przez zmianę wartości pod danym adresem.

## 7.6 PROC Zero

cel: zeruje blok pamięci

format: PROC Zero (BYTE POINTER adres, CARD rozmiar)

parametry: adres – jest wskaźnikiem do początkowego adresu bloku pamięci, który ma zostać wyzerowany  
rozmiar – jest wielkością zerowanego bloku

komentarz: procedura ta ustawia wszystkie wartości danego bloku pamięci na zero. Zerowany obszar znajduje się w przedziale: „adres” i „adres” + „rozmiar”-1.



## 7.7 PROC SetBlock

cel: zapełnia daną wartością cały blok pamięci

format: PROC SetBlock (BYTE POINTER adres, CARD rozmiar, BYTE wartość)

parametry: adres – wskaźnik początkowego adresu zapełnianego bloku  
rozmiar – wielkość zapełnianego bloku  
wartość – wartość, która będzie umieszczona w bajtach tego bloku

komentarz: procedura, ta ustawia wszystkie bajty danego bloku pamięci na wartość podaną w parametrze. Zapełniany obszar określają: „adres” i „adres”+”rozmiar”-1.

## 7.8 PROC MoveBlock

cel: przesuwa zawartość bloku pamięci

format: PROC MoveBlock (BYTE POINTER blok\_doc, blok\_źródł, CARD rozmiar)

parametry: blok\_doc – wskaźnik do początku bloku docelowego  
blok\_źródł – wskaźnik do początkowego adresu bloku źródłowego  
rozmiar – wielkość przesuwanego bloku

komentarz: procedura ta przesuwa wartości bloku źródłowego (począwszy od adresu „blok\_źródł” do adresu „blok\_źródł”+”rozmiar”-1) do bloku docelowego (począwszy od adresu „blok\_doc” do adresu „blok\_doc”+”rozmiar”-1). Jeżeli adres „blok\_doc” jest większy niż „blok\_źródł” i między nimi jest obszar mniejszy niż określony przez „rozmiar”, procedura nie będzie działać poprawnie ponieważ część bloku źródłowego będzie zachodzić na blok docelowy.

## 7.9 BYTE device

„device” jest zmienną zdefiniowaną w bibliotece *ACTION!* i pozwala sterować w operacjach wejścia/wyjścia „kanałem specjalnym” (default).

Ze zmiennej tej korzysta większość podprogramów wejścia/wyjścia. Po rozpoczęciu pracy na komputerze jest ona ustawiona na „0” co oznacza, że podprogramy takie jak Input, Put, Print zapisują lub odczytują dane z ekranu. Zmieniając wartość zmiennej „device” spowoduje się, że podprogramy te będą działać na innym urządzeniu.

Poniższe instrukcje służą do przydzielenia kanału specjalnego do drukarki:

```
Close(5) ; dla uniknięcia błędu „plik już otwarty”
Open(5, „P:”, 8)
device=5
```

Aby z powrotem przydzielić kanał specjalny do ekranu należy wykonać:

```
Close(5) ; zamknięcie „P:”
device=0
```

## 7.10 BYTE TRACE

Ta zmienna biblioteczna pozwala sterować opcją „TRACE” kompilatora z poziomu programu użytkownika. Należy w tym celu ustawić ją na żadaną wartość dyrektywą „SET”. Musi to być zrobione na początku programu. Ustawienie „TRACE” na 0 powoduje, że opcja ta staje się nieaktywna, a na 1, że będzie aktywna.

Przykład: SET TRACE = 0

## 7.11 BYTE LIST

Zmienna biblioteczna „LIST” steruje opcją kompilatora „LIST” w podobny sposób jak zmienna „TRACE”.

## 7.12 BYTE ARRAY EOF(8)

Ta zmienna biblioteczna pozwala zorientować się czy na danym kanale został osiągnięty koniec pliku (*End of File*). Np. jeżeli chce się sprawdzić czy w kanale 1 został „osiągnięty koniec pliku należy użyć następujących instrukcji:

```
IF EOF(1) THEN
    ; coś tu wpisz
    ...
FI
```

EOF wynosi 1 jeżeli został osiągnięty koniec pliku, w przeciwnym wypadku ma wartość 0.

# DODATKI

## Dodatek A      Składnia języka *ACTION!*

Poniżej zamieszczamy składnię języka *ACTION!* w notacji Backusa-Naura. Wytłumaczenia wymagają:

<u>symbol</u>	<u>znaczenie</u>
::=	„jest zdefiniowany jako”
	„lub”
[ ]	„opcjonalne”

Dodatek ma następującą budowę:

<b>A.1</b>	Stałe .....	104
	Stałe numeryczne	
	Stałe tekstowe	
	Stałe kompilacji	
<b>A.2</b>	Operatory i podstawowe typy danych .....	104
	Operatory	
	Podstawowe typy danych	
<b>A.3.</b>	Struktura programu w <i>ACTION!</i> .....	104
	Program <i>ACTION!</i>	
<b>A.4.</b>	Deklaracje .....	104
	Deklaracje systemowe	
	Dyrektywa DEFINE	
	Deklaracja TYPE (dla rekordów)	
	Deklaracje zmiennych	
	Deklaracja zmiennych dla podstawowych typów danych	
	Deklaracja zmiennych wskaźnikowych	
	Deklaracja tablic	
	Deklaracja rekordów	
<b>A.5</b>	Odwołania do zmiennych .....	105
	Odwołania do pamięci	
	Odwołania do zmiennych podstawowych typów	
	Odwołania do zmiennych wskaźnikowych	
	Odwołania do tablic	
	Odwołania do rekordów	
<b>A.6</b>	Podprogramy <i>ACTION!</i> .....	105
	Podprogramy	
	Struktura procedury	
	Struktura funkcji	
	Wywołanie podprogramu	
	Parametry	
<b>A.7</b>	Instrukcje .....	106
	Instrukcja przypisania	
	Instrukcja EXIT	
	Instrukcja IF	
	Pętla DO-OD	
	Instrukcja UNTIL	
	Pętla WHILE	
	Pętla FOR	
	Bloki kodowe	
<b>A.8</b>	Wyrażenia .....	106
	Wyrażenia logiczne	
	Wyrażenia arytmetyczne	

## A.1 Stałe ACTION

### Stałe numeryczne

```

<num const> ::= <dec num> | <hex num> | <char>
<dec num> ::= <dec num> <digit> | <digit>
<hex num> ::= <hex num> <hex digit> | $<hex digit>
<char> ::= <any printable character>
<hex digit> ::= <digit> | A | B | C | D | E | F
<digit> ::= 0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

### Stałe tekstowe

```

<str const> ::= „<string>„
<string> ::= <string> <str char> | <str char>
<str char> ::= <all printable characters except ">

```

### Stałe kompilacji

```

<comp const> ::= <comp const> + <base comp const> | <base comp const>
<base comp const> ::= <identifier> | <num const> | <ptr ref>

```

## A.2 Operatory i podstawowe typy danych

### Operatory

```

<special op> ::= AND | OR | & | %
<rel op> ::= XOR | ! | = | # | <> | < | <= | > | >=
<add op> ::= + | -
<mult op> ::= * | / | MOD | LSH | RSH
<unary op> ::= @ | -

```

### Podstawowe typy danych

```

<fund type> ::= CARD | CHAR | BYTE | INT

```

## A.3. Struktura programu w ACTION!

### Program ACTION!

```

<program> ::= <program> MODULE <prog module> | [MODULE] <prog module>
<prog module> ::= [<system decls>] <routine list>

```

## A.4. Deklaracje

### Deklaracje systemowe

```

<system decls> ::= <DEFINE decl> | <TYPE decl> | <var decl>

```

### Dyrektywa DEFINE

```

<DEFINE decl> ::= <DEFINE> <def list>
<def list> ::= <def list>, <def> | <def>
<def> ::= <identifier> = <str const>

```

### Deklaracja TYPE (dla rekordów)

```

<TYPE decl> ::= TYPE <rec ident list>
<rec ident list> ::= <rec ident list> <rec ident> | <rec ident>
<rec ident> ::= <rec name> = [<field init>]
<rec name> ::= <identifier>
<field init> ::= <fund var decl>

```

### Deklaracje zmiennych

```

<var decl> ::= <var decl> <base var decl> | <base var decl>
<base var decl> ::= <fund decl> | <POINTER decl> | <ARRAY decl> | <record decl>

```

### Deklaracja zmiennych dla podstawowych typów danych

```

<fund decl> ::= <fund decl> <base fund decl> | <base fund decl>
<base fund decl> ::= <fund type> <fund ident list>
<fund type> ::= CARD | CHAR | BYTE | INT
<fund ident list> ::= <fund ident list>, <fund ident> | <fund ident>

```

```

<fund ident> ::= <identifier> [=<init opts>]
<init opts> ::= <addr> | [<value>]
<addr> ::= <comp const>
<value> ::= <num const>

```

### Deklaracja zmiennych wskaźnikowych

```

<POINTER decl> ::= <ptr type> POINTER <ptr ident list>
<ptr type> ::= <fund type> | <rec name>
<ptr ident list> ::= <ptr ident list>, <ptr ident> | <ptr ident>
<ptr ident> ::= <identifier> [=<value>]

```

### Deklaracja tablic

```

<ARRAY decl> ::= <fund type> ARRAY <arr ident list>
<arr ident list> ::= <arr ident list>, <arr ident> | <arr ident>
<arr ident> ::= <identifier> [( <dim> )] [=<arr init opts>] <dim> ::= <num const>
<arr init opts> ::= <addr> | [<value>] | <str const>
<addr> ::= <comp const>
<value list> ::= <value list> <value> | <value>
<value> ::= <comp const>

```

### Deklaracja rekordów

```

<record decl> ::= <identifier> <rec ident list>
<rec ident list> ::= <rec ident list>, <rec ident> | <rec ident>
<rec ident> ::= <identifier> [=<address>]
<address> ::= <comp const>

```

## A.5 Odwołania do zmiennych

### Odwołania do pamięci

```

<mem reference> ::= <mem contents> | @<identifier>
<mem contents> ::= <fund ref> | <arr ref> | <ptr ref> | <rec ref>
<fund ref> ::= <identifier>
<arr ref> ::= <identifier> (<arith exp>)
<ptr ref> ::= <identifier>^
<rec ref> ::= <identifier>.<identifier>

```

## A.6 Podprogramy *ACTION!*

### Podprogramy

```

<routine list> ::= <routine list> <routine> | <routine>
<routine> ::= <proc routine> | <func routine>

```

### Struktura procedury

```

<proc routine> ::= < <PROC decl> [<system decls>] [<stmt list>] [RETURN]
<PROC decl> ::= PROC <identifier> [=<addr>] ([<param decl>])
<addr> ::= <comp const>

```

### Struktura funkcji

```

<func routine> ::= <FUNC decl> [<system decl>] [<stmt list>] [RETURN (<arith exp>)]
<FUNC decl> ::= <fund type> FUNC <identifier> [=<addr>] ([<param decl>])
<addr> ::= <comp const>

```

### Wywołanie podprogramu

```

<routine call> ::= <FUNC call> | <PROC call>
<FUNC call> ::= <identifier> ([<params>])
<PROC call> ::= <identifier>([<params>])

```

### Parametry

```

<param decl> ::= <var decl>      UWAGA: dozwolone max. 8 parametrów

```

## A.7 Instrukcje

### Instrukcje

```

<stmt list> ::= <stmt list> <stmt> | <stmt>
<stmt> ::= <simp stmt> | <struc stmt> | <code block>
<simp stmt> ::= <assign stmt> | <EXIT stmt> | <routine call>
<struc stmt> ::= <IF stmt> | <DO loop> | <WHILE loop> | <FOR loop>

```

### Instrukcja przypisania

```

<assign stmt> ::= <mem contents> = <arith exp>

```

### Instrukcja EXIT

```

<EXIT stmt> ::= EXIT

```

### Instrukcja IF

```

<IF stmt> ::= IF <cond exp> THEN (stmt list) [[:ELSEIF extern :]] [[:ELSE exten]] FI
<ELSEIF exten> ::= ELSEIF <cond exp> THEN [<stmt list>]
<ELSE exten> ::= ELSE [<stmt list>]

```

### Pętla DO-OD

```

<DO loop> ::= DO [<stmt list>] [<UNTIL stmt>] OD

```

### Instrukcja UNTIL

```

<UNTIL stmt> ::= UNTIL <cond exp>

```

### Pętla WHILE

```

<WHILE loop> ::= WHILE <cond exp> <DO loop>

```

### Pętla FOR

```

<FOR loop> ::= FOR <identifier> = <start> TO <finish> [STEP <inc>] <DO loop>
<start> ::= <arith exp>
<finish> ::= <arith exp>
<inc> ::= <arith exp>

```

### Bloki kodowe

```

<code block> ::= [<comp const list>]
<comp const list> ::= <comp const list> <comp const> | <comp const>

```

## A.8 Wyrażenia

### Wyrażenia logiczne

```

<complex rel> ::= <complex rel> <special op> <simp rel exp> | <simp rel exp> <special op> <simp rel exp>
<simple rel exp> ::= <arith exp> <rel op> <arith exp>

```

### Wyrażenia arytmetyczne

```

<arith exp> ::= <arith exp> <add op> <mult exp> | <mult exp>
<mult exp> ::= <mult exp> <mult op> <value> | <value>
<value> ::= <num const> | <mem reference> | (<arith exp>)

```

**Dodatek B****Mapa pamięci *ACTION!***

\$00 - \$C9	zmienna systemu operacyjnego i <i>ACTION!</i>
\$CA - \$CD	obszar wolny
\$CE - \$D3	zmienne <i>ACTION!</i>
\$D4 - \$FF	rejstry operacji zmiennoprzecinkowych Atari
\$100 - \$3FF	system operacyjny
\$400 - \$4FF	zmienne <i>ACTION!</i>
\$500 - \$5FF	bufor Atari dla operacji zmiennoprzecinkowych Atari
\$600 - MEMLO-1	system operacyjny
MEMLO	stos kompilatora <i>ACTION!</i>
MEMLO+\$200	bufor edytora <i>ACTION!</i>
MEMLO+\$300	Tablice „hash” <i>ACTION!</i>
LO+\$750	bufor tekstowy edytora <i>ACTION!</i>
	obszar generowanego kodu przez kompilator <i>ACTION!</i>
TOP - \$800	tablica symboli kompilatora <i>ACTION!</i>
MEMTOP	pamięć ekranu
\$A000	<i>ACTION!</i> Cartridge
\$C000 - \$FFFF	system operacyjny, pamięć ROM itd.

**UWAGA:** Obszar przeznaczony na kod programu wygenerowany przez kompilator zaczyna-się w miejscu gdzie kończy się bufor tekstowy edytora. Zob. część V, rozdział 2.

## Dodatek C                      Komunikaty błędów

W dodatku tym opiszemy znaczenie wszystkich błędów, które mogą się pojawić podczas programowania w *ACTION!*. Przedstawiamy tylko te błędy, które wykrywa system *ACTION!* i pomijamy błędy wykrywane przez system operacyjny (błędy 128-255).

Numer błędu	Komentarz
<b>0</b>	niewystarczająca pamięć. W części II §4.3 i w części V §4.4 opisaliśmy, jakie należy podjąć kroki w przypadku pojawienia się tego błędu
<b>1</b>	opuszczony „,,,,” (cudzysłów) w łańcuchu tekstowym
<b>2</b>	zagnieżdżone dyrektywy DEFINE. Jest to zabronione
<b>3</b>	przepełniona tablica symboli zmiennych globalnych
<b>4</b>	przepełniona tablica symboli zmiennych lokalnych
<b>5</b>	błąd składni dyrektywy SET
<b>6</b>	błędny format deklaracji
<b>7</b>	niepoprawna lista argumentów. Zbyt duża liczba argumentów w instrukcji lub podprogramie
<b>8</b>	zmienna niezadeklarowana. Zanim zmienna zostanie użyta musi być wcześniej zadeklarowana
<b>9</b>	użycie zmiennej w miejscu gdzie powinna być stała
<b>10</b>	nielegalne przypisanie, np. Var = 5>7 jest niepoprawna
<b>11</b>	błąd nierozpoznany
<b>12</b>	opuszczone THEN
<b>13</b>	opuszczone FI
<b>14</b>	przepełniony obszar na kod programu. Zob. część V §4.4
<b>15</b>	opuszczone DO
<b>16</b>	opuszczone TO
<b>17</b>	niepoprawny format wyrażenia
<b>18</b>	niezgodność nawiasów
<b>19</b>	opuszczone OD
<b>20</b>	niemożność przydzielenia pamięci
<b>21</b>	błędne odwołanie się do tablicy
<b>22</b>	plik wejściowy jest zbyt duży. Należy podzielić go na mniejsze części
<b>23</b>	niepoprawne wyrażenie warunkowe
<b>24</b>	niepoprawna składnia instrukcji FOR
<b>25</b>	nielegalne EXIT. Nie znajduje się w pętli DO-OD
<b>26</b>	zbyt dużo zagnieżdżeń (dozwolone max. 16 poziomów)
<b>27</b>	niepoprawna składnia TYPE
<b>28</b>	nielegalne RETURN
<b>61</b>	przekroczenie tablicy symboli. Zob. część V.
<b>128</b>	program został zatrzymany poprzez naciśnięcie klawisza BREAK



**Dodatek D****Bibliografia**

ATARI Personal Computer System Operating System User^ Manual ana Hardware Manual

ATARI 810 Disk Drice Operator”s Manual

ATARI 400/800 Disk Utility

ATARI 400/800 Operating Systems

ATARI 400/800 Disk Operating Systems II Reference Manual

OSS OS/A+ reference manual

OSS DOS XL reference manual

oraz dodatkowo:

Poole, McNiff, Cook, Your Atari Computer



## E.10 Odszukiwanie i wstawianie tekstu

- szukanie łańcucha <CTRL> <SHIFT> F, wprowadź tekst
- podstawianie <CTRL> <SHIFT> S, wprowadź nowy tekst, <RETURN>, wprowadź stary łańcuch

## E.11 Dzielenie i łączenie linii

- dzielenie linii ustaw kursor, <CTRL> <SHIFT> <RETURN>
- łączenie linii ustaw kursor na początku drugiej linii, <CTRL> <SHIFT> <BACK S>

## E.12 Wyjście z edytora

- wyjście z edytora <CTRL> <SHIFT> M

**Dodatek F****Zbiór komend monitora *ACTION!***

B	restart systemu <i>ACTION!</i>
C {„<specyfikacja pliku>„}	kompilacja programu
D	wywołanie DOS
E	przejsie na poziom edytora
O	wybór opcji <i>ACTION!</i>
F	kontynuacja zatrzymanego programu
R {„<specyfikacja pliku>„}	uruchomienie programu <i>ACTION!</i>
SET <adres> = <wartość>	ustawia w danym miejscu pamięci zadaną wartość
W {„<specyfikacja pliku>„}	zapamiętanie skompilowanego programu na dysku
X <instrukcja>  :, <instrukcja> :	wykonanie instrukcji języka
? <adres>	wyświetla wartość określoną przez adres
* < adres>	wyświetla wartości pamięci począwszy od zadanego adresu

## Dodatek G                      Menu opcji

Pytanie	Standardowo	Zakres
<b>Display on?</b> Steruje ekranem podczas kompilacji i operacji wejścia/wyjścia	Y	Y lub N
<b>Bell off?</b> Steruje sygnałem dźwiękowym	N	Y lub N
<b>Case insensitive?</b> „N” oznacza, że kompilator nie będzie sprawdzał czy słowa kluczowe języka są napisane dużymi literami, a zmienne napisane innymi rodzajem liter będą traktowane jako te same.	N	Y lub N
<b>Trace on?</b> „Y” oznacza, że kompilator spowoduje, że w czasie wykonywania programu będą wyświetlane wszystkie wejścia do procedur lub funkcji.	N	Y lub N
<b>List on?</b> Steruje listowaniem programu na ekran podczas procesu kompilacji	N	Y lub N
<b>Window size?</b> Ustawia rozmiar okna tekstowego nr 1. Okno 1 i okno 2 łącznie mogą zajmować 23 linie.	18	5 do 18
<b>Line size?</b> Steruje długością linii.	120	1 do 240
<b>Left margin?</b> Ustawia lewy margines okna tekstowego.	2	0 do 39
<b>EOL character?</b> Zmienia znak końca linii na inny (możliwy do zobaczenia).	\$9B	dowolny znak ATASCII

## Dodatek H „PRIMES” Benchmark

Nie został do tej pory opracowany sposób na porównywanie względem siebie różnych komputerów i języków programowania. Można jedynie badać czas wykonywania pewnego typu działań. Służą do tego specjalne programy testujące tzw. Benchmark. Poniżej przedstawiamy taki program zamieszczony w magazynie BYTE z września 1981 r. Przedstawiamy również nasze czasy, aby można je było porównać z podanymi w czasopiśmie.

<u>tryb</u>	<u>czas</u>
kompilacja	≈ 25 sek.
wyłączony ekran	12.2 sek.
włączony ekran	17.9 sek.

```

DEFINE size = „8190”,
        OK = „1”,
        OFF = “0”

BYTE ARRAY flags(size+1)
CARD count, i, k, prime
BYTE DISPLAY = $22F,
        iter,
        tick = 20,
        tock = 19

PROC Primes()
  DISPLAY = 0
  tick = 0
  tock = 0

  FOR iter =1 TO 10
    DO
      count = 0
      SetBlock (flags, size, ON)
      FOR i = 0 TO size
        DO
          IF flags (i) THEN
            prime = i+i+3
            ; PrintCE(prime)
            k = prime + 1
            WHILE k <= size
              DO
                flags(k) = OFF
                k ==+ prime
              OD
            count ==+ 1
          FI
        OD
      OD
      i = tick+256+tock
      DISPLAY = $22
      Printf (n%0 Primes done in U% ticks %E”,, count, i;
RETURN

```

## Dodatek I Przenoszenie instrukcji BASIC do programu ACTION!

W dodatku tym przedstawiamy niektóre funkcje. Podprogramy, instrukcje itd. z języka programowania BASIC. Dla każdej z nich pokazujemy odpowiednik w języku ACTION!

W przykładach z języka BASIC nie podajemy numeru linii ponieważ jest on tutaj zbędny. Należy jednak pamiętać, że w programach w tym języku linie muszą być numerowane.

Dla przykładów z języka ACTION! przyjęto następujące deklaracje zmiennych:

```
INT i, j, k
CARD c, d, e
BYTE a, b
BYTE ARRAY s, t, aa, ba
CARD ARRAY ca, da, ea
INTARRAY ia, ja, ka
```

Instrukcje BASIC	Odpowiednik w ACTION!
C = D+I*A	c = d + i * a
10 IF A<>0 THEN30 20 B=1:C=A*3 30 REM	IF a<>0 THEN b=1:c=a*3 FI
10 IF A=0 THEN B=1: GOTO 30 20 B=7 30 REM	IF a<>0 THEN b=1 ELSE B=7 FI
FOR I=1 TO 100 ...	FOR i=1 to 100 DO ...
NEXT I	OD
PRINT "HELLO"	PrintE(„HELLO“)
PRINT „HELLO“;	Print („HELLO“ )
PRINT #5;„HELLO“	PrintDE(5,„HELLO“)
PRINT #5";": " HELLO";	PrintD(5,„HELLO“ )
PRINT I	PrintIE(i)
PRINT "I=„; I	PrintF("I=%I%E", i) lub Print(„I=„) PrintIE(i)
PRINT #3; B*3;	PrintBD (3, b*3)
INPUT I	Put („?) : i =InputI() Zwróćcie uwagę na opcjonalny „:“ (dwukropek) w przykładzie ACTION! Dwukropki są ignorowane: przez ACTION! i dlatego też mogą być używane jako separatory instrukcji.
INPUT	B\$Put(„?) : InputS (ba)
PUT #0, 65	Put („A) lub Put (65) lub Put (\$41)
GET #C, B	b = GetD(c)
OPEN #1, 4, 0, „K:“	Open (1, „K:“, 4, 0)
CLOSE #3	Close(3)
NOTE #1, C, B	Note(1, @c, @b)

Instrukcje BASIC	Odpowiednik w ACTION!
POINT #1, C, B	Point (1, c, b)
XIO 18, #6, 0, 0, „S:”	XIO (6, 0, 18, 0, 0, „S:”) albo podprogram Fill
B=PEEK(C)	b = Peek(c) lub ba = c : b = ba^
POKE C, B	Poke(c, b) lub ba = c : ba^ = b
GRAPHICS 8	Graphics (8)
COLOR 3	color = 3 <b>UWAGA:</b> color jest zmienną zdefiniowaną przez ACTION!
DRAWTO C, D	DrawTo (c, d)
LOCATE C, D, B	b= Locate (c, d)
PLOT C, D	Plot (c, d)
POSITION C, D	Position (c, d)
SETCOLOR 0, 1, C	SetColor (0, 1, c)
GRAPHICS 24 : COLOR C PLOT 200, 150 DRAWTO 120, 20 POSITION 40, 150 POKE 765, C XIO 18, #6, 0, 0, „S:”	Graphics (24) : color = c Plot (200, 150) DrawTo (120, 20) Fill (40, 150)
SOUND 0, 121, 10, 6	Sound (0, 121, 10, 6)
C = PADDLE(b) C = PTRIG(B) C = STICK(B) C = STRIG(B)	c = Paddle (b) c = Ptrig (b) c = Stick (b) c = Strig (b)
B\$ = S\$	SCopy(ba, a)
B\$ = S\$(3, 5)	SCopyS(ba, s, 3, 5 )
B\$(3,5) = S\$	SAssign(ba, s, 3, 5)
B=INT(6*RND (0))+1	b = Rand (6) + 1
FOR C = 4000 TO 5000 : POKE C,0 : NEXT C	Zero (4000, 1001)
STOP	Break ()
B\$ = STR4 (I)	StrI (i, ba)
I = VAL (S4 )	i = ValI (s)



# **ACTION!**

## **The Best Complete Software Development System**

**The Fastest, High Level Language Available for the Atari\*:** *A versatile, structured language that runs at almost assembly language speeds (100 + times faster than BASIC).*

**Best Structured Language:** *incorporates features found in PASCAL, C, ALGOL, and ADA, yet has many of the same commands familiar to Atari BASIC programmers.*

### **Has Everything You Need:**

**THE EDITOR:** *Many advanced features for easily creating and modifying source text... two separate program windows, each allowing up to 240 characters per line... fast horizontal and vertical scrolling... move and copy text... string find and replace... and much more!*

**THE MONITOR:** *Selects compilation options, saves compiled programs, examines variable values and memory locations... and even traces the execution of your programs.*

**THE COMPILER:** *Super fast compilation into machine code, accepting source from the Editor or from tape or disk.*

**THE LIBRARY:** *A built in collection of useful subroutines for you to use in your programs including: string manipulation... print procedures and formatting... I/O routines... and, graphics and game controller routines.*

# **ACTION!**

## **Najlepszy kompletny System do tworzenia oprogramowania**

**Najszybszy język wysokiego poziomu dostępny dla Atari\*:** wszechstronny, język strukturalny, który działa niemal równie szybko jak asembler (100 x razy szybciej niż BASIC).

**Najlepszy język strukturalny:** zawiera funkcje dostępne w językach PASCAL, C, ALGOL i ADA, a także zawiera wiele tych samych poleceń, które są znane programistom Atari BASIC.

### **Ma wszystko, czego potrzebujesz:**

**EDYTOR:** wiele zaawansowanych funkcji umożliwiających łatwe tworzenie i modyfikowanie tekstu źródłowego... dwa osobne okna programu, każde mieszczące do 240 znaków w linii... szybkie przewijanie w poziomie i pionie... przenoszenie i kopiowanie tekstu... wyszukiwanie ciągów znaków wymieniść... i wiele więcej!

**MONITOR:** wybiera opcje kompilacji, zapisuje skompilowane programy, sprawdza wartości zmiennych i lokalizacje pamięci... a nawet śledzi wykonywanie programów.

**KOMPILATOR:** super szybka kompilacja do kodu maszynowego, akceptująca źródło z Edytora lub z taśmy czy dysku.

**BIBLIOTEKA:** wbudowany zbiór przydatnych podprogramów do wykorzystania w programach, w tym: manipulowanie ciągami znaków... procedury drukowania i formatowanie... procedury we/wy... oraz procedury graficzne i kontrolera gier.