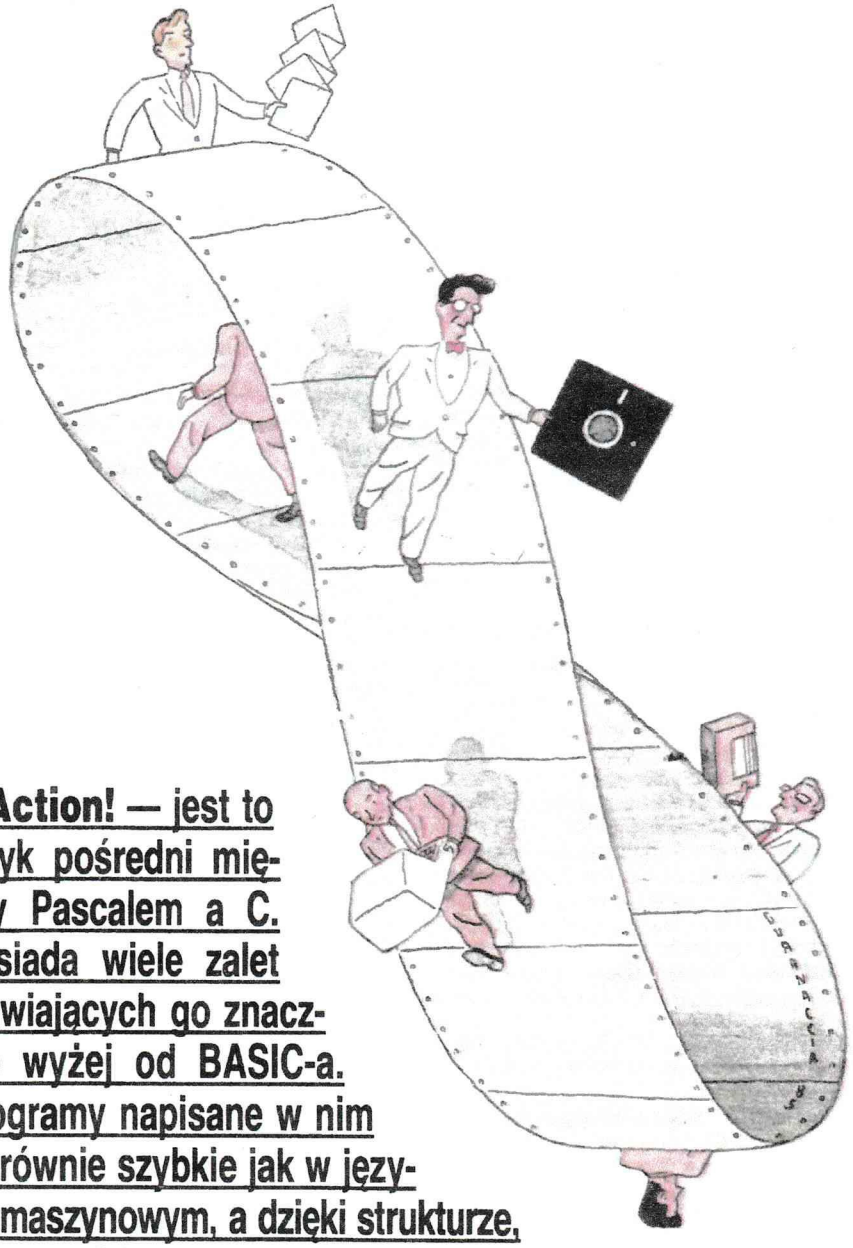
Action!

NOWY JĘZYK ATARI



Action! — jest to język pośredni między Pascalem a C. Posiada wiele zalet stawiających go znacznie wyżej od BASIC-a. Programy napisane w nim są równie szybkie jak w języku maszynowym, a dzięki strukturze, edytorowi i bogatej bibliotece procedur programowanie jest znacznie łatwiejsze. Kompilator Action! jest dostępny na wszystkich trzech nośnikach stosowanych w Atari: na kasecie, na dysku i na cartridge'u. Oprócz tego w serwisie Atari można wbudować Action! na stałe do komputera.

System **Action**! składa się z edytora i kompilatora oraz kontrolującego pracę całości monitora. Program źródłowy piszemy przy pomocy edytora, a następnie kompilujemy. Możliwe jest wykonanie napisanego programu zarówno po skompilowaniu, jak i w wersji źródłowej. W drugim przypadku monitor wczytuje program źródłowy, kompiluje go, a następnie wykonuje. Wadą dostępnego w Polsce systemu **Action**! jest konieczność jego wczytania przed uruchomieniem programu (również skompilowanego). Wynika to z faktu, że kompilator nie dołącza do programu wynikowego biblioteki procedur. Można oczywiście nabyć i taki kompilator, lecz jedynie u producenta.

Programy w **Action**! budowane są ze składników, z których każdy zawiera grupę odpowiednich instrukcji wykonujących jakąś operację. Dzięki temu można łatwo dołączać do pisanego programu wcześniej napisane bloki. Jedynym wymaganiem takiego strukturalnego podejścia jest to, że program musi być złożony z prawidłowych składników. Składnikami tymi są w **Action**! procedury i funkcje. Zwykle program zawiera ich wiele, lecz wymagany jest co najmniej jeden. W programie zawierającym kilka procedur wykonywanie rozpoczynane jest od ostatniej, tak więc powinna ona sterować całym programem.

Zanim zaczniemy naukę programowania w **Action**! musimy zapoznać się z działaniem poszczególnych elementów systemu.

EDYTOR

Edytor służy do tworzenia nowych programów i redagowania starych. Edytor **Action**! jest jednym z najlepiej rozwiązanych edytorów programowych. Można go używać również do innych celów (programowanie w innych językach, redagowania tekstów itp.).

Po uruchomieniu systemu w dolnej części ekranu pojawia się jasna linia z napisem „ACTION! (c)1983 ACS". Jest to linia komunikatów, w której wyświetlane są raporty błędów, inne informacje i wprowadzane komendy edytora. Gdy używasz dwóch okien edytora, linia komunikatów umieszczona jest między oknami.

Edytor **Action**! jest zaprogramowany w sposób umożliwiający jak najłatwiejsze pisanie programów. Pozwala on na wpisanie linii o długości do 240 znaków, pomimo, że w oknie edytora widoczne jest tylko 38 znaków. Dzięki temu można pisać programy bardziej przejrzyście. Osiągnięcie maksymalnej dopuszczalnej długości linii edytor sygnalizuje dźwiękiem buczka. Jeżeli wpisana linia jest dłuższa niż szerokość okna, to znak na krawędzi okna (lewej lub prawej) jest wyświetlany w negatywie (inverse video), aby uwidocznić ten fakt.

Tekst programu wpisuje się do edytora bez żadnych specjalnych poleceń — jak przy użyciu maszyny do pisania. Aby wpisać znak kontrolny należy najpierw nacisnąć klawisz ESC. Mówi to edytorowi, że następny znak ma być interpretowany jako

tekst, a nie jako polecenie dla edytora. Poleceniami edytora są niektóre kombinacje klawiszy naciskanych równocześnie, np. **SHIFT-CLEAR**.

TRYBY EDYTORA

Edytor może pracować w dwóch trybach: wstawiania i zastępowania. W trybie zastępowania wprowadzany tekst zastępuje tekst już istniejący, znak po znaku. W trybie wstawiania nowy tekst jest wprowadzany w środek starego bez jego kasowania. Po uruchomieniu edytor pracuje w trybie zastępowania. Do przełączania trybów pracy edytora służy polecenie **SHIFT-CONTROL-I**.

OBSŁUGA EDYTORA

Aby usunąć tekst z edytora należy użyć polecenia **SHIFT-CLEAR**. Kasuje on nie tylko tekst widoczny na ekranie, lecz całą zawartość edytora. Gdy używane są dwa okna edytora, to polecenie dotyczy tylko tego okna, w którym aktualnie znajduje się kursor. Aby uniknąć omyłkowego zniszczenia redagowanego programu, edytor pyta: „Clear?". Należy odpowiedzieć „Y" (tak) lub „N" (nie). Jeśli w redagowanym tekście dokonano zmian i nie został on zapisany, edytor pyta ponownie: „Not saved-Delete?" (Nie zapisane, Usunąć?), aby upewnić się, że nie chcesz zapisać poprawionej wersji.

Istnieje tylko jeden sposób opuszczenia edytora (oczywiście oprócz wyłączenia komputera) — **SHIFT-CONTROL-M**. Polecenie to powoduje przekazanie kontroli systemu do monitora, skąd można wywołać inne elementy systemu lub przejść do DOS-u.

Program napisany w edytorze może być zapisany na dowolnym urządzeniu zewnętrznym. Umożliwia to polecenie **SHIFT-CONTROL-W**. Po naciśnięciu tej kombinacji klawiszy pojawia się napis „Write?". Przy zapisywaniu pliku na dysku należy podać pełną nazwę pliku — D: nazwa.ext. Przy zapisie na magnetofonie lub drukarce wystarczy nazwa urządzenia — odpowiednio C: lub P:.

Analogicznie odczytu zapisanego pliku dokonuje się poleceniem **SHIFT-CONTROL-R,** po którym komputer pyta „Read?". Trzeba teraz podać nazwę pliku w sposób wyżej opisany. Dodatkowo polecenie to umożliwia odczyt spisu zawartości dyskietki (directory). Następuje to po podaniu jako nazwy pliku tekstu ?1:*.*. Gdy chcemy odczytać directory z innej stacji dysków, to wystarczy cyfrę 1 zastąpić numerem tej stacji.

RUCH KURSORA

Poniższe polecenia umożliwiają poruszanie kursora w całym obszarze edytora **Action**!:

CONTROL-←	przesunięcie kursora o jeden znak w lewo;
CONTROL-→	przesunięcie kursora o jeden znak w prawo;
CONTROL-↑	przesunięcie kursora o jedną linię w górę;
CONTROL-↓	przesunięcie kursora o jedną linię w dół;
SHIFT-CONTROL-<	przesunięcie kursora na początek aktualnej linii;
SHIFT-CONTROL->	przesunięcie kursora na koniec aktualnej linii;
SHIFT-CONTROL-H	przesunięcie kursora na początek pliku;
TAB	przesunięcie kursora do następnej pozycji tabulacji;
SHIFT-TAB	ustawienie pozycji tabulacji;
CONTROL-TAB	skasowanie pozycji tabulacji.

Czasami zachodzi potrzeba podzielenia linii programu na dwie linie. W tym celu należy ustawić kursor na znaku, który ma być pierwszym znakiem w drugiej linii i nacisnąć **SHIFT-CONTROL-RETURN**. Odwrotna operacja jest wykonywana po ustawieniu kursora na pierwszym znaku drugiej linii i naciśnięciu klawiszy **SHIFT-CONTROL-BACKSPACE**.

REDAGOWANIE TEKSTU

Edytor **Action**! posiada większość funkcji stosowanych w normalnych edytorach tekstu. Jedną z nich jest zamiana pewnych fragmentów tekstu na inne. Umożliwia to polecenie **SHIFT-CONTROL-S**. Po jego użyciu wyświetlany jest napis „Substitute?" i jeśli funkcja ta była już używana, to wyświetlany jest również ostatnio wprowadzony „nowy" tekst. Należy wpisać tekst, który ma być umieszczony w redagowanym programie i nacisnąć RETURN, a samo naciśnięcie RETURN zachowuje poprzedni tekst. Teraz wyświetlany jest napis „for?", po którym trzeba wpisać tekst, który ma być zastąpiony (gdy funkcja była już używana należy postąpić analogicznie jak z „nowym" tekstem). Po naciśnięciu RETURN edytor znajduje pierwsze wystąpienie podanego „starego" tekstu i zamienia go na „nowy". Następne polecenie **SHIFT-CONTROL-S** powoduje zamianę wskazanego tekstu w następnym miejscu występowania bez pytań „Substitute?" i „for?" (jeżeli nie została przed tym użyta inna funkcja edytora). Jeżeli podany „stary" tekst nie występuje w redagowanym programie, to wyświetlany jest komunikat „not found" (nie znaleziony).

Możliwe jest także przemieszczanie lub kopiowanie całych bloków tekstu poprzez bufor kopiowania. Za każdym razem, gdy użyte zostanie polecenie **SHIFT-DELETE**, usunięta linia jest umieszczana w buforze kopiowania. Polecenie **SHIFT-CONTROL-P** wstawia usuniętą linię na aktualnej pozycji kursora. Bufor kopiowania jest kasowany przy każdym użyciu **SHIFT-DELETE**, lecz z jednym wyjątkiem. Jeśli **SHIFT-DELETE** jest używane raz po raz (bez innych poleceń lub wpisywania tekstu), to bufor nie jest kasowany. Pozwala to na umieszczenie w buforze całego bloku tekstu i jego wstawienie w dowolnym miejscu poprzez **SHIFT-CONTROL-P**. Ponieważ wstawianie tekstu nie kasuje bufora, to można go kopiować kilkakrotnie.

Jeżeli podczas redagowania linii programu zrobiony został błąd, to można odtworzyć pierwotną postać tej linii poleceniem **SHIFT-CONTROL-U**. Funkcja ta działa tylko pod warunkiem, że kursor nie opuszczał odtwarzanej linii.

OKNA

Dodatkową funkcją oferowaną przez edytor **Action**! jest możliwość utworzenia drugiego okna i jednoczesnego redagowania dwóch różnych programów. Drugie okno jest tworzone poleceniem **SHIFT-CONTROL-2**. Polecenie to służy także do przeniesienia kursora z pierwszego do drugiego okna. Przeniesienie kursora z okna drugiego do pierwszego wykonuje się poleceniem **SHIFT-CONTROL-1**. W celu skasowania okna najpierw należy umieścić w nim kursor, a następnie nacisnąć klawisze **SHIFT-CONTROL-D**. W linii komunikatów pojawia się napis „Delete Window?". Dalsze postępowanie jest takie samo jak przy kasowaniu zawartości edytora.

Okno może być przesuwane w górę i w dół przez ruch kursora. Dodatkowe polecenia umożliwiają przesunięcie okna w całości w górę lub w dół. Po poleceniu **SHIFT-CONTROL-↑** okno przemieszcza się w górę tak, że najwyższa linia starego okna jest teraz linią najniższą. Analogicznie polecenie **SHIFT-CONTROL-↓** przesuwa okno w dół. Możliwe jest również przesunięcie całego okna o jeden znak w prawo (**SHIFT-CONTROL-]**) lub w lewo (**SHIFT-CONTROL-[**).

ETYKIETY

Etykiety pozwalają oznaczyć dowolne miejsce w tekście. W celu ustawienia etykiety na aktualnej pozycji kursora trzeba podać polecenie **SHIFT-CONTROL-T** i po wyświetleniu pytania „tag id:" wpisać jeden znak identyfikujący etykietę i nacisnąć RETURN. Jeżeli etykieta oznaczona podanym znakiem już istnieje, to jest ona usuwana i ustawiana w nowym miejscu.

Polecenie **SHIFT-CONTROL-G** przenosi kursor do etykiety, której znak zostanie wpisany w odpowiedzi na pytanie „tag id:". Gdy podanej etykiety nie ma w tekście, wyświetlany jest komunikat „tag not set" (etykieta nie ustawiona).

UWAGA. Każda operacja, która zmienia zawartość linii, kasuje etykiety w tej linii.

MONITOR

Monitor **Action**! steruje pracą całego systemu. Linia poleceń monitora znajduje się w górnej części ekranu i zawiera znak > oraz kursor. Pozostała część ekranu stanowi obszar komunikatów. Ma on wiele zastosowań. W czasie działania programu służy do wyświetlania jego wyników. Może być także wykorzystany do śledzenia wykonywania programu. Gdy w programie zostanie wykryty błąd, to w obszarze komunikatów wyświetlany jest numer błędu i część programu z linią zawierającą błąd.

POLECENIA MONITORA

Monitor rozpoznaje podane mu polecenia tylko po pierwszej literze, nie trzeba więc wpisywać całych słów a tylko literę początkową. Wykonanie wpisanego polecenia następuje po naciśnięciu RETURN.

BOOT (B) — Powoduje restart systemu i powrót do edytora. Kasuje wszystkie zawarte w pamięci programy i zmienne.

COMPILE (C) — Program napisany w **Action**! musi być skomplikowany przed wykonaniem. Polecenie COMPILE wywołuje kompilator **Action**! i powoduje skompilowanie programu znajdującego się w pamięci edytora. Można także skompilować program zapisany na urządzeniu zewnętrznym. W tym celu trzeba podać nazwę, np C "C:" dla magnetofonu lub C "D1: nazwa.ext" dla stacji dysków. Gdy program ma być odczytany ze stacji numer 1, to można opuścić nazwę urządzenia: C "nazwa.ext". Jeżeli kompilator napotka w programie błąd, to przerywa kompilację i przekazuje sterowanie ponownie do monitora.

DOS (D) — Opuszczenie systemu **Action**! i przejście do DOS-u (zawartość edytora i monitora zostaje zniszczona).

EDITOR (E) — Przejście do edytora. Jeżeli w czasie kompilacji programu wystąpił błąd, to po przejściu do edytora kursor znajdzie się w linii następującej po błędzie.

OPTIONS (O) — Wywołanie menu opcji, które pozwala na zmianę parametrów pracy edytora, kompilatora i monitora. Każda opcja jest wyświetlana w linii poleceń. Jeżeli chcesz ją zmienić, wpisz nową wartość i naciśnij RETURN. Samo naciśnięcie RETURN pozostawia daną opcję bez zmian. Menu opcji można opuścić naciskając ESC.

Poniżej wymienione są wszystkie dostępne opcje. Format opisu: nazwa opcji — tłumaczenie (wartość standardowa — części systemu, na które wpływa).

Display? — Ekran (Y — E,K,M)
Ekran może być wyłączany w celu zwiększenia szybkości kompilacji i operacji wejścia/wyjścia. Wpisanie „N" (nie) wyłącza ekran, wpisanie „Y" (tak) pozostawia go włączony cały czas.

Bell? — Buczek (Y — E,K,M)
Komputer daje dźwięk buczka zawsze, gdy napotkany zostanie błąd. Można go wyłączyć przez wpisanie tu „N", wpisanie „Y" włącza buczek ponownie.

Case sensitive? — Rozróżnianie liter (N — K)
Gdy ta opcja jest ustawiona na „Y", to rozróżniane są małe i duże litery (np. „suma", „Suma" i „SUMA" są różne), a instrukcje języka (np. FOR,

WHILE itd.) muszą być pisane dużymi literami. Jednakże dla ułatwienia pracy początkującym programistom ta opcja jest wyłączona (N) po uruchomieniu systemu.

Trace? — Śledzenie (N — K)
Dzięki tej opcji można śledzić kompilację programu. Gdy jest ona włączona (Y), kompilator wyświetla w obszarze komunikatów wszystkie wywoływane procedury wraz z ich parametrami.

List? — Listing (N — K)
Ustawienie tej opcji na „Y" powoduje wyświetlanie w obszarze komunikatów aktualnie kompilowanej linii.

Window 1 size? — Rozmiar okna 1 (18 — E)
Rozmiar okna 1 w edytorze **Action**! jest ustalany bezpośrednio. Wielkość okna 2 jest ustalana pośrednio w zależności od okna 1 — oba okna razem muszą mieć 23 wiersze. Gdy mamy dwa okna, każde z nich może zawierać nie mniej niż 5 wierszy i nie więcej niż 18 wierszy. Wpisanie wartości spoza tego przedziału spowoduje dopasowanie jej do dopuszczalnej wartości.

Line size? — Rozmiar linii (120 — E)
Rozmiar linii jest liczbą znaków w tej linii. Maksymalny rozmiar linii wynosi 240 znaków. Opcja ta pozwala kontrolować długość linii przy wydruku programu na drukarce.

Left margin? — Lewy margines (2 — E,M)
Lewy margines jest miejsce, od którego wyświetlane są znaki na ekranie. Może mieć wartość z zakresu od 0 do 39.

EOL character? — Znak końca linii (spacja — E)
Znak końca linii jest wyświetlany przez edytor **Action**! na końcu każdej linii. Normalnie jest on niewidoczny (pusta spacja), a uwidocznienie go może pomóc w redagowaniu programu. Może tu wpisać dowolny znak (także graficzny) — sugeruję użycie CONTROL-. lub CONTROL–T.

PROCEED (P) — Wznawia wykonywanie programu zatrzymanego naciśnięciem klawisza BREAK lub przy użyciu procedury 'Break'. Program jest kontynuowany tak jakby nie nastąpiło żadne przerwanie.

RUN (R) — Uruchomienie skompilowanego programu. Może występować w czterech różnych formatach:

RUN	— uruchomienie programu znajdującego się w pamięci edytora;
RUN "nazwa pliku"	— wczytanie i uruchomienie programu z urządzenia zewnętrznego;
RUN adres	— uruchomienie programu od podanego adresu;
RUN nazwa procedury	— uruchomienie jednej procedury z programu lub biblioteki procedur.

SET (S) — Polecenie umożliwiające bezpośredni dostęp do pamięci RAM i zmianę jej zawartości. Działa tak samo jak instrukcja SET w języku **Action**!.

WRITE (W) — Pozwala na zapisanie skompilowanego programu na urządzeniu zewnętrznym jako pliku binarnego. Program zapisany w ten sposób na dysku może być uruchomiony bezpośrednio z DOS-u (po wczytaniu kompilatora). Przy zapisie na kasecie należy podać nazwę "C:", a na dysku — "D1:nazwa.ext" (dla stacji numer 1 D1: można opuścić).

XECUTE (X) — Polecenie umożliwiające wykonanie dowolnej instrukcji języka **Action**! lub dowolnej dyrektywy kompilatora (oprócz MODULE i SET) z monitora. Po poleceniu należy wpisać instrukcję do wykonania, np. XECUTE PrintE("Witaj w Action!").

? — Wyświetlenie aktualnej zawartości pamięci o podanym adresie. Adresem może być dowolna stała kompilatora (opisane są w dalszej części). W obszarze komunikatów wyświetlany jest adres w postaci dziesiętnej i szesnastkowej, znak ATASCII o kodzie odpowiadającym zawartości komórki, zawartość dwóch komórek w postaci szesnastkowej oraz liczby dziesiętne w formacie BYTE i CARD. Na przykład:

```
? $FFFE
65535,$FFFE = s $E6F3 243 59123
```

***** — Wyświetlenie aktualnej zawartości pamięci od podanego adresu. Adresem może być dowolna stała kompilatora. Format wyświetlanego wyniku jest taki jak opisany wyżej. Wyświetlanie przerywa się naciśnięciem klawisza spacji. Naciśnięcie **CONTROL–1** chwilowo wstrzymuje wyświetlanie, a ponowne naciśnięcie wznawia je.

URUCHAMIANIE PROGRAMU

Przy uruchamianiu napisanego programu często zachodzi konieczność wyszukiwania popełnionych błędów. Dzięki możliwościom monitora **Action**! jest to czynność łatwa i niekłopotliwa.

Opcja TRACE — Po zezwoleniu tej opcji monitora kompilowany i wykonywany program jest śledzony na ekranie. Nazwa każdej wywołanej procedury jest wyświetlana na ekranie wraz z przekazywanymi parametrami.

Przed wprowadzeniem do uruchamianego programu poprawek konieczne jest zatrzymanie jego wykonywania. W **Action**! jest to możliwe dwoma sposobami: klawiszem BREAK i procedurą 'Break'.

Klawisz BREAK — Zasadniczo klawisz BREAK nie działa w **Action**! Jednak w dwóch przypadkach można przy jego pomocy przerwać wykonywany program: podczas operacji wejścia/wyjścia i przy wywoływaniu procedur, które mają więcej niż 3 parametry.

Procedura biblioteczna PROC Break() — Jeżeli chcemy zatrzymywać uruchamiany program co jakiś czas w celu sprawdzenia poprawności jego działania, to należy w kilku miejscach umieścić wywołanie procedury 'Break'. Działa ona jak instrukcja BASIC-a STOP, a działanie programu można wznowić poleceniem PROCEED.

SŁOWA KLUCZOWE

Bliższą znajomość z językiem **Action**! zaczniemy od poznania „słów kluczowych". Są to słowa i symbole, które mają specjalne znaczenie dla kompilatora — instrukcje, deklaracje, operatory i dyrektywy kompilatora. Słowa kluczowe mogą być używane tylko w sposób zdefiniowany w **Action**!, w szczególności nie wolno ich używać jako nazw zmiennych. Oto lista tych słów:

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

SYMBOLE

Przy opisie języka będziemy używać pewnych symboli dla uwidocznienia składni. Oto lista tych symboli wraz z objaśnieniem:

adres — Adres jest to numer komórki w pamięci komputera. Gdy polecamy komputerowi, aby umieścił coś w pamięci, to musimy podać mu adres (jak na poczcie).

identyfikator — Jest to nazwa określająca zmienną, procedurę itp. Nazwa w **Action**! musi spełniać następujące warunki:
1. Musi rozpoczynać się od litery.
2. Pozostałe znaki muszą być literami, cyframi lub znakiem podkreślenia (_).
3. Nie może być słowem kluczowym.

litera — Dowolna litera alfabetu, zarówno mała, jak i duża.

znak — Litera lub cyfra od 0 do 9.

MSB, LSB — MSB oznacza Most Significant Byte (bardziej znaczący bajt), LSB — Least Significant Byte (mniej znaczący bajt). W systemie dziesiętnym nie ma bajtów znaczących, lecz cyfry. Na przykład w liczbie 54 bardziej znaczącą cyfrą jest 5, a mniej — 4. Podobnie jest w zapisie komputerowym, ale zgodnie ze standardem mikroprocesora 6502 liczby są zapamiętywane i używane w kolejności LSB, MSB.

$ — Znak dolara, gdy jest użyty przed liczbą, oznacza liczbę w zapisie szesnastkowym (hexadecymalnym).

; — Średnik jest symbolem komentarza i wszystko, co po nim znajduje się w linii jest ignorowane przez kompilator.

< i > — Wyrażenie objęte tymi symbolami określa, co powinno znaleźć się w tym miejscu w instrukcji lub linii programu. Np. <identyfikator> oznacza, że musi być użyty prawidłowy identyfikator.
Wyrażenie zawarte między tymi symbolami może być użyte, lecz nie musi. Np. [<identyfikator>] oznacza, że można użyć identyfikatora, ale nie jest to konieczne.
Te symbole oznaczają powtórzenie (jak w muzyce). Np. I:<identyfikator>:I oznacza, że można użyć dowolnej liczby identyfikatorów (również żadnego)
Ten symbol oznacza alternatywę. Np. wyrażenie <identyfikator> *< adres> należy odczytać: użyj identyfikatora lub adresu, ale nie obu na raz.

TYPY DANYCH

Przed opisem typów danych stosowanych w **Action**! musimy trochę powiedzieć o zmiennych i stałych, ponieważ są one podstawowymi obiektami, na których operuje komputer.

ZMIENNE

Dozwolona nazwa zmiennej musi być prawidłowym identyfikatorem. Nie ma żadnych innych ograniczeń dla stosowanych nazw. Aby uzyskać przejrzysty program należy używać znaczących nazw zmiennych np. 'licznik' zamiast 'i'. Nie wpływa to na długość programu wynikowego (skompilowanego), a jedynie zwiększa o kilka bajtów program źródłowy.

STAŁE

W **Action**! używane są trzy typy stałych: stałe liczbowe, stałe znakowe i stałe kompilatora.

Stałe liczbowe mogą być wprowadzane w trzech różnych formach: jako liczby dziesiętne, liczby szesnastkowe i znaki.

Liczby dziesiętne nie wymagają żadnego specjalnego określenia, np. 2, 46, 324, 65500, -4360 (ujemna). Liczby szesnastkowe są oznaczane znakiem dolara ($) przed liczbą, np. $0D, $300, $4A00, -$8C (ujemna). Stałe znakowe są oznaczane apostrofem (') przed znakiem, np. 'A, '@, '", 'f.

Znaki są stałymi liczbowymi, ponieważ są one wewnętrznie reprezentowane przez liczbę jednobajtową, która jest równa kodowi ATASCII znaku.

Stałe kompilatora różnią się od trzech powyższych typów. Są one używane w czasie kompilacji programu od ustawienia pewnych atrybutów zmiennych, procedur oraz funkcji i nie są stosowane podczas działania programu. Stałe kompilatora mogą mieć następujące formaty:

1. stała liczbowa
2. identyfikator
3. wskaźnik
4. suma dwóch dowolnych wyżej wymienionych

Pierwszy format nie wymaga wyjaśnień, lecz pozostałe trzeba krótko opisać. Gdy używasz identyfikatora (nazwy zmiennej, procedury lub funkcji) jako stałej kompilatora, to używaną wartością jest adres tego identyfikatora. Trzeci format pozwala użyć jako stałych kompilatora wskaźników tablic. Ostatni jest zwykłym dodawaniem dowolnych dwóch z trzech pierwszych typów. Dla lepszego zrozumienia podamy kilka przykładów:

```
licznik     ;adres zmiennej "licznik"
$8D00       ;stała szesnastkowa
numer^      ;wskaźnik jako stała
5+wsk^      ;5 plus zawartość wskaźnika "wsk"
$80+n       ;$80 plus adres zmiennej "n"
```

TYPY ZMIENNYCH

W **Action!** występują trzy podstawowe typy zmiennych: **BYTE**, **CARD** i **INT** oraz dodatkowe, które będą opisane później. Wszystkie podstawowe typy zmiennych są liczbowe.

BYTE — Dodatnie liczby całkowite mniejsze niż 256. Są one reprezentowane wewnętrznie przez jednobajtową liczbę bez znaku (z zakresu od 0 do 255). Dzięki takiej reprezentacji BYTE może być używana jako zmienna znakowa, można więc używać zamiennie słów kluczowych **BYTE** i **CHAR**.

CARD — Podobnie jak **BYTE**, lecz o większym zakresie. Reprezentacją wewnętrzną jest liczba dwubajtowa, więc zmienne typu **CARD** mogą mieć wartość z zakresu od 0 do 65535. Liczby typu **CARD** są zapamiętywane w formacie LSB, MSB.

INT — Liczby całkowite ze znakiem, reprezentacja i zapis jak **CARD**. Najstarszy bit jest interpretowany jako znak, więc zakres liczb **INT** jest od -32768 do 32767.

DEKLARACJE

Przed użyciem każdego identyfikatora trzeba go zadeklarować, aby komputer zarezerwował odpowiednie miejsce w pamięci. Format deklaracji jest następujący:

<typ><identyf.> [=<inform.pocz.>]|,<identyf.> [=<inform.pocz.>] ...|

gdzie <inform.pocz.> jest początkową wartością lub adresem zmiennej: <adres> I[<wartość>]

Przykłady:

```
BYTE wynik,ilosc    ;deklaracja zmiennych "wynik" i "ilosc"
                    ;jako typ BYTE
INT num=[0]         ;zmienna "num" typu INT o wartości 0
BYTE x=$8000        ;zmienna "x" typu BYTE umieszczona w pa-
                    ;mięci pod adresem $8000
CARD pkt=[0],       ;zmienne CARD "pkt" o wartości 0 i "wsk"
wsk=$83D4           ;pod adresem $83D4. Zmienne nie muszą
                    ;być w tej samej linii
```

Deklaracje zmiennych muszą znajdować się w programie bezpośrednio po instrukcji MODULE albo na początku procedury lub funkcji.

Wojciech Zientara

POLECENIA EDYTORA

Przejście do monitora	‹CTRL›‹SHIFT› M (monitor)

Polecenia I/O

Odczyt pliku	‹CTRL›‹SHIFT› R — nazwa (read)
Odczyt directory	‹CTRL›‹SHIFT› R — ?n:✱.✱ (read)
Zapis pliku	‹CTRL›‹SHIFT› W — nazwa (write)
Wydruk	‹CTRL›‹SHIFT› W — P: (write)

Ruch kursora

Kursor w górę	‹CTRL› ↑
Kursor w dół	‹CTRL› ↓
Kursor w prawo	‹CTRL› →
Kursor w lewo	‹CTRL›←
Kursor na początek linii	‹CTRL›‹SHIFT› <
Kursor na koniec linii	‹CTRL›‹SHIFT› >
Następna linia	‹RETURN›
Tabulacja	‹TAB›
Ustawienie tabulacji	‹SHIFT›‹TAB›
Skasowanie tabulacji	‹CTRL›‹TAB›

Ruch okna ekranu

Początek pliku	‹CTRL›‹SHIFT› H (head)
1 ekran w górę	‹CTRL›‹SHIFT› ↑
1 ekran w dół	‹CTRL›‹SHIFT› ↓
1 znak w lewo	‹CTRL›‹SHIFT›]
1 znak w prawo	‹CTRL›‹SHIFT› [
Utworzenie drugiego okna	‹CTRL›‹SHIFT› 2
Przejście do pierwszego okna	‹CTRL›‹SHIFT› 1
Przejście do drugiego okna	‹CTRL›‹SHIFT› 2
Kasowanie okna	‹CTRL›‹SHIFT› D (delete)

Redagowanie tekstu

Wstawianie/wymiana	‹CTRL›‹SHIFT› I (insert)
Odtworzenie zmienionej linii	‹CTRL›‹SHIFT› U (undone)
Odtworzenie usuniętej linii	‹CTRL›‹SHIFT› P (paste)
Zapamiętanie bloku tekstu	‹SHIFT›‹DELETE›
Wstawienie tekstu z bufora	‹CTRL›‹SHIFT› P (paste)
Wyszukiwanie łańcucha	‹CTRL›‹SHIFT› F (find)
Wymiana łańcucha	‹CTRL›‹SHIFT› S (substitute)
Podzielenie linii na dwie	‹CTRL›‹SHIFT›‹RETURN›
Połączenie dwóch linii	‹CTRL›‹SHIFT›‹DELETE›
Ustawienie etykiety	‹CTRL›‹SHIFT› T (tag set)
Odszukanie etykiety	‹CTRL›‹SHIFT› G (go to tag)

POLECENIA MONITORA

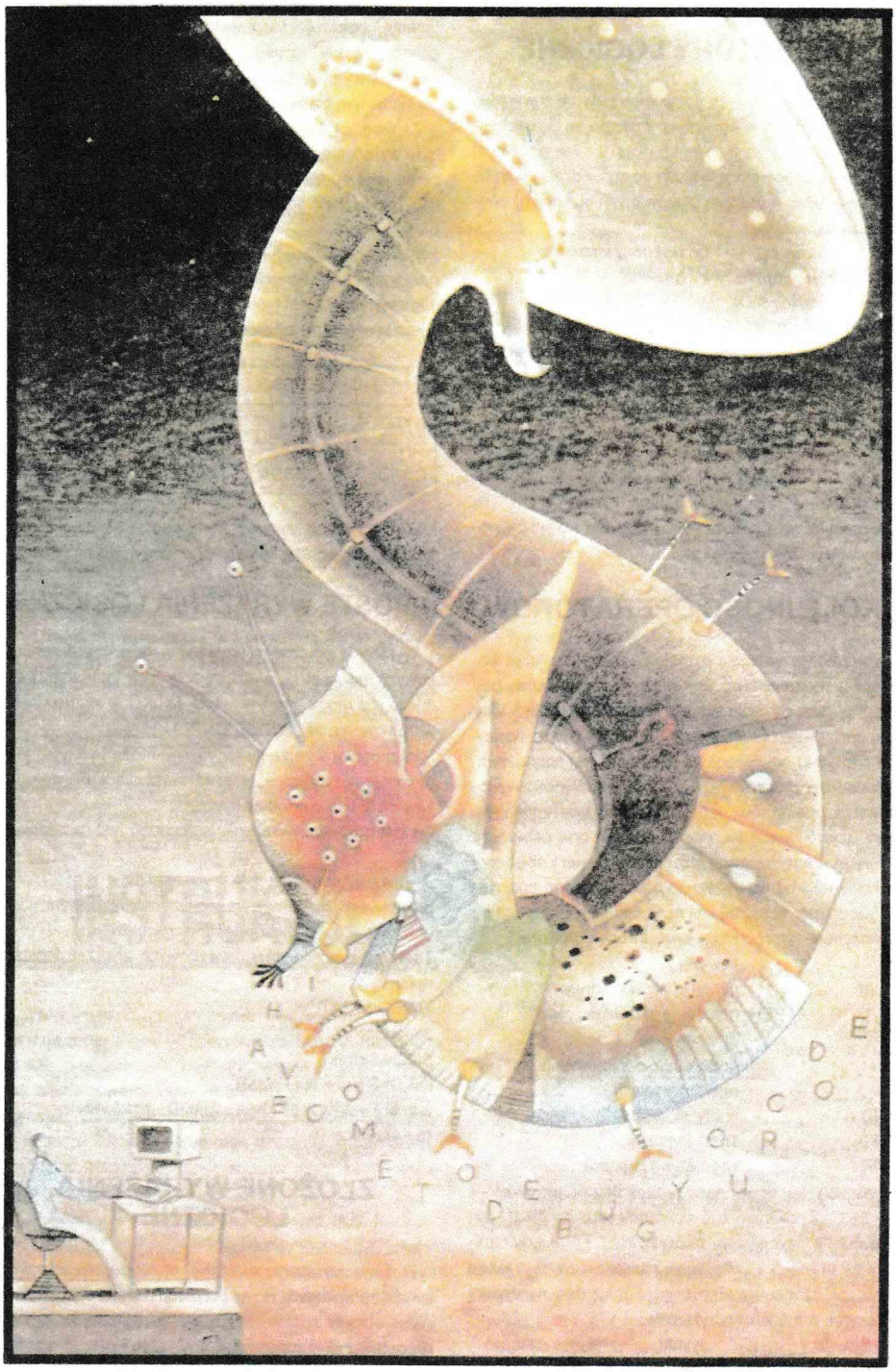
Restart Action!	B (boot)
Przejście do DOS-u	D (DOS)
Przejście do edytora	E (editor)
Menu wariantów	O (options)
Wykonanie instrukcji	X ‹instrukcja› (xecute)
Kompilacja programu	C ["nazwa"] (compile)
Zapis programu	W ["nazwa"] (write)
Uruchomienie programu	R ["nazwa"] (run)
Kontynuacja programu	P (proceed)
Ustawienie wartości (POKE)	SET ‹adres› = ‹wartosc›
Zawartość komórki (PEEK)	? ‹adres›
Przegląd pamięci	✱ ‹adres›

KODY BŁĘDÓW Action!

Kod	Znaczenie
0	Za mały obszar dostępnej pamięci.
1	Brak cudzysłowu (") w łańcuchu.
2	Zagnieżdżona dyrektywa DEFINE.
3	Brak miejsca w tablicy symboli zmiennych globalnych.
4	Brak miejsca w tablicy symboli zmiennych lokalnych.
5	Błąd składni w dyrektywie SET.
6	Nieprawidłowy format deklaracji.
7	Zbyt dużo argumentów w instrukcji lub procedurze.
8	Niezadeklarowana zmienna.
9	Użyto zmienną w miejscu, w którym wymagana jest stała.
10	Niedozwolona instrukcja podstawienia.
11	Nierozpoznany błąd.
12	Brak instrukcji THEN.
13	Brak instrukcji FI.
14	Zbyt mały obszar pamięci dla kodu wynikowego.
15	Brak instrukcji DO.
16	Brak instrukcji TO.
17	Nieprawidłowy format wyrażenia.
18	Niezamknięty nawias.
19	Brak instrukcji OD.
20	Nie można przesunąć więcej pamięci.
21	Niedozwolona tablica.
22	Zbyt duży plik wejściowy.
23	Niedozwolone wyrażenie warunkowe.
24	Błąd składni w instrukcji FOR.
25	Niedozwolona instrukcja EXIT.
26	Zbyt głębokie zagnieżdżenie (dozwolone 16 poziomów).
27	Błąd składni w instrukcji TYPE.
28	Niedozwolona instrukcja RETURN.
61	Brak miejsca w tablicy symboli.
128	Zatrzymanie programu klawiszem BREAK.

Action!

—(2)—



Zamieszczony w poprzednim wydaniu specjalnym o Atari wstępny opis języka Action! spotkał się z ogromnym zainteresowaniem Czytelników. Będziemy więc kontynuować kurs programowania w tym języku, a równolegle w normalnych „Bajtkach" będą publikowane krótkie programy przykładowe.

WYRAŻENIA

Wyrażenia są konstrukcjami, które uzyskują wartość ze zmiennych, stałych i warunków za pomocą specjalnego zestawu operatorów. Na przykład „4+3" jest wyrazeniem, które jest równe „7" dopóki znakiem „+" oznaczamy dodawanie. Jeżeli jako operatora użyjemy znaku „*" (oznaczającego mnożenie), to wyrażenie będzie równe „12" (4*3=12). W **Action!** występują dwa typy wyrażeń: arytmetyczne i logiczne. W powyższych przykładach zostały użyte wyrażenia arytmetyczne. Wyrażeniem logicznym jest takie, które w wyniku daje odpowiedź „prawda" lub „fałsz". „5›=7" jest fałszywe, jeśli „›=" oznacza „większe lub równe". Takie wyrażenia są używane w instrukcjach warunkowych. Instrukcją warunkową stosowaną codziennie jest na przykład zdanie: „Jeżeli jest czwarta lub później, to czas kończyć pracę".

Przed szczegółowym omówieniem wyrażeń musimy zdefiniować operatory, które mogą wystąpić w tych wyrażeniach.

OPERATORY

W **Action!** stosuje się trzy rodzaje operatorów:

1. operatory arytmetyczne
2. operatory bitowe
3. operatory logiczne

Jak sugerują nazwy, pierwsze i ostatnie operatory odnoszą się do odpowiednich typów wyrażeń. Operatory bitowe służą również do operacji arytmetycznych, lecz wykonywanych na poszczególnych bitach liczb.

OPERATORY ARYTMETYCZNE

Operatorami arytmetycznymi są te, których używamy w matematyce, lecz nieco zmodyfikowane, aby umożliwić wprowadzenie ich z klawiatury komputera. Oto lista operatorów dostępnych w **Action!** wraz z opisem:

- – minus znakowy, oznacza liczbę ujemną, np. –5
- * mnożenie, np. 4*3
- / dzielenie całkowite (bez reszty), np. 13/5 (jest to równe 2, gdyż reszta jest pomijana)
- MOD reszta z dzielenia całkowitego, np. 13 MOD 5 (jest to równe 3, gdyż 13/5 = 2 z resztą 3)
- \+ dodawanie, np. 4+3
- – odejmowanie, np. 4–3

Zwróć uwagę, że znak „=" nie jest operatorem arytmetycznym. Jest on używany tylko w wyrażeniach logicznych, instrukcjach przypisania i niektórych deklaracjach.

PROGRAMOWANIE

OPERATORY BITOWE

Operatory bitowe działają na liczbach w postaci dwójkowej (binarnej). Oznacza to, że możesz wykonywać operacje podobne do wykonywanych przez komputer (ponieważ on zawsze pracuje na liczbach dwójkowych). Poniższa lista zawiera wszystkie dostępne operatory bitowe:

&	bitowy iloczyn logiczny — AND
%	bitowa suma logiczna — OR
!	bitowa alternatywa logiczna — EXCLUSIVE-OR
XOR	to samo co „!"
LSH	przesunięcie w lewo
RSH	przesunięcie w prawo
@	adres

Pierwsze trzy operatory porównują liczby bit po bicie i zwracają wynik zależny od rodzaju operatora, jak pokazano niżej.

& porównuje dwa bity zwracając wartość według tabelki:

bit A	bit B	wynik
1	1	1
0	1	0
1	0	0
0	0	0

```
Przykład: 5 & 39 — 00000101 (równe dziesiętnie 5)
                   00100111 (równe dziesiętnie 39)
                   & ----------
                   00000101 (wynikiem & jest 5)
```

% porównuje dwa bity zwracając wartość według tabelki:

bit A	bit B	wynik
1	1	1
0	1	1
1	0	1
0	0	0

```
Przykład: 5 % 39 -- 00000101 (5)
                    00100111 (39)
                    % ----------
                    00100111 (wynikiem % jest 39)
```

! porównuje dwa bity zwracając wartość według tabelki:

bit A	bit B	wynik
1	1	0
0	1	1
1	0	1
0	0	0

```
Przykład: 5 ! 39 -- 00000101 (5)
                    00100111 (39)
                    ! ----------
                    00100010 (wynikiem ! jest 34)
```

Zarówno LSH jak i RSH przesuwają bity. Jeżeli operują na liczbach dwubajtowych (CARD i INT), to działają na oba bajty jednocześnie. W przypadku liczby typu INT może przy tym nastąpić zmiana znaku. Format użycia jest następujący:

‹operand› ‹operator› ‹liczba przesunięć›

gdzie ‹operand› jest stałą lub zmienną liczbową,
‹operator› LSH lub RSH
‹liczba przesunięć› stała lub zmienna liczbowa określająca liczbę bitów do przesunięcia

Najlepiej działanie tych operatorów zilustrują przykłady:

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

operacja	MSB	LSB
———	01010110	11001010 ($56CA)
LSH 1	10101101	10010100 ($56CA LSH 1 = $AD94)
RSH 1	00101011	01100101 ($56CA RSH 1 = $2B65)
LSH 2	01011011	00101000 ($56CA LSH 2 = $5B28)
RSH 2	00010101	10110010 ($56CA RSH 2 = $15B2)

Łatwo zauważyć, że LSH 1 jest równe mnożeniu przez dwa, RSH 1 — dzieleniu przez 2 (dla liczb dodatnich). W rzeczywistości ten sposób mnożenia i dzielenia przez 2 jest szybszy niż użycie "*2" i "/2", ponieważ komputer nie musi już tłumaczyć wyrażenia na formę dwójkową.

Operator "@" daje w wyniku adres zmiennej stojącej po prawej stronie operatora. Nie może on być używany ze stałymi. "@ctr" zwraca adres, pod którym znajduje się w pamięci zmienna "ctr". Operator ten jest bardzo użyteczny przy działaniach na wskaźnikach.

OPERATORY LOGICZNE

Operatory logiczne są dozwolone jedynie w wyrażeniach logicznych, a wyrażenia logiczne są dozwolone tylko w instrukcjach IF, WHILE i UNTIL. Jak wcześniej wspomniano operatory te służą do sprawdzania warunków równości. A oto lista stosowanych w **Action!** operatorów logicznych:

=	równe, np. 4=7 (to jest oczywiście fałsz)
#	nierówne, np. 4#7 (prawda)
‹›	to samo co #
›	większe niż, np. 9›2 (prawda)
›=	większe lub równe, np. 5›=5 (to jest prawda)
‹	mniejsze niż, np. 2‹9 (prawda)
‹=	mniejsze lub równe, np. 5‹=5 (to jest prawda)
AND	iloczyn logiczny (I)
OR	suma logiczna (LUB)

Zarówno "#" jak i "‹›" oznaczają to samo, więc możesz używać tego, który wolisz. "AND" i "OR" są specjalnymi operatorami i będą opisane razem ze złożonymi wyrażeniami logicznymi.

KOLEJNOŚĆ OPERATORÓW

Operatory wymagają określenia priorytetu, czyli kolejności wykonywania, aby uniknąć przypadków, gdy nie wiadomo jak obliczyć wyrażenie, np. 4+5*3. Czy jest ono równe (4+5*)3, czy 4+(5*3) ? Bez znajomości priorytetu odpowiedź na to pytanie jest niemożliwa. **Action!** ma precyzyjnie wyznaczoną kolejność wykonywania operatorów, lecz może być ona zmieniana przy użyciu nawiasów, ponieważ mają one najwyższy priorytet. Zamieszczona niżej tabela zawiera operatory w kolejności od najwyższego do najniższego priorytetu. Operatory w jednej linii mają taki sam priorytet i są obliczane w wyrażeniu od lewej strony do prawej.

()	nawiasy
– @	minus znakowy, adres
* /MOD LSH RSH	mnożenie, dzielenie, reszta, przesunięcie
+–	dodawanie, odejmowanie
=# ‹› ›= ‹=	operatory relacji
AND &	iloczyn logiczny
OR %	suma logiczna
XOR !	alternatywa logiczna

Według tej tabeli nasz wcześniejszy przykład — 4+5*3 — powinien być obliczany jako 4+(5*3), ponieważ "*" ma wyższy priorytet niż "+". A gdy napiszemy (4+5)*3 ? Dołączenie nawiasów zmienia kolejność obliczania wyrażenia, gdyż mają one najwyższy priorytet. A oto kilka przykładów:

wyrażenie	wynik	kolejność obliczeń
4/2*3	6	/,*
43 MOD 7* 2 + 19	21	MOD,*,+
–((4+2)/3*	-2	+,/,–

WYRAŻENIA ARYTMETYCZNE

Dowolne wyrażenie arytmetyczne może składać się ze stałych i zmiennych liczbowych oraz operatorów porządkujących sposób uzyskania wyniku. Format wyrażenia jest następujący:

‹operand› ‹operator› ‹operand›

gdzie operand jest stałą liczbową, zmienną liczbową, wywołaniem funkcji (FUNC) lub innym wyrażeniem arytmetycznym. Pierwsze trzy możliwości są jasne, lecz czwarta może sprawić kłopot. Prosty przykład wyjaśni o co chodzi:

wyrażenie początkowe: 3*(4+(22/7)*2)

krok	wyrażenie	wynik	uproszczone wyrażenie
0	3*(4+(22/7)*2)	–	---
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

"krok" jest numerem wykonywanego obliczenia; "wyrażenie" pokazuje, które wyrażenie jest aktualnie obliczane; "wynik" zawiera wynik tego obliczenia; a „uproszczone wyrażenie" pokazuje wygląd wyrażenia po aktualnym obliczeniu.

Zauważ, że wyrażenia w krokach od 2 do 4 zawierają inne wyrażenie jako jeden z operandów, lecz to „wyrażenie jako operand" jest już obliczone i można w to miejsce wstawić liczbę jak w „uproszczonym wyrażeniu".

Wyrażenia arytmetyczne w **Action!** mogą mieć jako operandy dane różnych typów. Wynik takiego mieszanego wyrażenia jest typu nadrzędnego, jak widać w zamieszczonej niżej tabelce:

	BYTE	INT	CARD
BYTE	BYTE	INT	CARD
INT	INT	INT	CARD
CARD	CARDA	CARD	CARD

UWAGA: użycie minusa znakowego (—) daje wynik typu INT, a operator adresu @ daje wynik typu CARD.

PROSTE WYRAŻENIA LOGICZNE

Wyrażenia logiczne są używane w instrukcjach warunkowych do kontroli wykonywania tych instrukcji. Pamiętaj, że mogą one być użyte TYLKO w instrukcjach warunkowych (IF, WHILE, UNTIL).

W prostym wyrażeniu logicznym może być tylko jeden operator logiczny, więc sprawdzanie wielu warunków musi być wykonane inaczej.

Format prostego wyrażenia logicznego jest następujący:

‹wyr arytm› ‹oper log› ‹wyr arytm›

gdzie ‹wyr arytm› jest wyrażeniem arytmetycznym
‹oper log› jest operatorem logicznym

Teraz kilka przykładów prawidłowych wyrażeń logicznych:

```
@kot‹=$22A7
zmienna›'y
5932#licznik
(5&7)*8›=(3*(kot+pies))
adr/$FF+(@wsk+indeks)‹›$FU3D–wsk&indeks
(5+4)*9›licz–1
```

ZŁOŻONE WYRAŻENIA LOGICZNE

Złożone wyrażenia logiczne pozwalają rozszerzyć zakres sprawdzania przez połączenie wielu warunków. Jeżeli chcesz coś zrobić tylko w lipcową niedzielę, to jak przekazać komputerowi, aby sprawdził, czy jest lipiec, a potem, czy jest niedziela. W **Action!** służą do tego operatory logiczne AND i OR. Kompilator traktuje je jako specjalne oparatory do sprawdzania warunków złożonych z prostych wyrażeń logicznych. Ich format jest następujący:

‹wyr log› ‹oper sp› ‹wyr log› 1: ‹oper sp› ‹wyr log›:1

gdzie ‹wyr log› jest prostym wyrażeniem logicznym
‹oper sp› jest specjalnym operatorem logicznym (AND lub OR)

UWAGA: ten format nie ma wyjątków. Jeżeli spróbujesz napisać inaczej, to uzyskasz błąd kompilacji „Bad Expression" (złe wyrażenie).

Poniższa tablica prawdy pokazuje, jak każdy z tych operatorów działa w różnych sytuacjach. „wyr 1" i

wyrażenia		wyniki	
wyr 1	wyr 2	AND	OR
prawda	prawda	prawda	prawda
prawda	fałsz	fałsz	prawda
fałsz	prawda	fałsz	prawda
fałsz	fałsz	fałsz	fałsz

„wyr 2" są prostymi wyrażeniami logicznymi z obu stron operatora specjalnego; „prawda" i „fałsz" to możliwe wyniki wyrażenia logicznego.

UWAGA: można użyć nawiasów do zmiany kolejności obliczania wyrażenia logicznego. Jeśli tego nie zrobisz, to wyrażenie będzie obliczane według priorytetów operatorów.

I znów kilka przykładów prawidłowych złożonych wyrażeń logicznych:

```
kot<=5 AND pies<>13
(@wsk+7) * 3 # $60FF AND @wsk <= $1FFF
x!$F0>0 OR pies>=100
(8&kot)<10 OR @wsk<>$0D
kot<>0 AND (pies>400 OR pies<-400)
wsk=$D456 OR wsk=$E000 OR wsk=$600
```

Teraz trochę dziwna sytuacja:

```
$F0 AND $0F
```

jest to fałszywe, ponieważ „AND" zostanie potraktowane jako operator bitowy w wyrażeniu arytmetycznym, podczas gdy

```
$F0<>0 AND $0F<>0
```

jest prawdziwe, ponieważ „AND" rozdziela dwa proste wyrażenia logiczne, a więc jest traktowany jako specjalny operator w wyrażeniu złożonym.

UWAGA: operatory w parach „AND"–„&" i „OR"–„%" są sobie równoważne i kompilator zależnie od kontekstu traktuje je jako bitowe lub logiczne. Pożądane jest jednak przyzwyczajenie się do używanej tu konwencji stosowania „&" i „%" jako operatorów bitowych, a „AND" i „OR" jako logicznych, ze względu na zgodność z ewentualnymi wersjami rozwojowymi systemu **Action!**.

INSTRUKCJE

Program komputerowy będzie nieużyteczny, jeśli nie będzie mógł aktywnie operować danymi. Potrafisz już deklarować zmienne, stałe itd., lecz nie jest to sposób na manipulowanie nimi. Aktywną częścią każdego języka komputerowego są instrukcje i **Action!** nie jest tu wyjątkiem. Instrukcje tłumaczą działania, które chcesz wykonać, na formę, którą komputer może zrozumieć i prawidłowo wykonać.

W **Action!** występują dwa rodzaje instrukcji: instrukcje proste i instrukcje strukturalne. Instrukcje proste nie zawierają w sobie innych instrukcji, zaś instrukcje strukturalne są złożone z innych instrukcji (zarówno prostych, jak i strukturalnych) umieszczonych w pewnej określonej kolejności. Instrukcje strukturalne można jeszcze podzielić na instrukcje warunkowe i instrukcje pętli. Każda z tych kategorii będzie omówiona osobno.

INSTRUKCJE PROSTE

Instrukcje proste są to te, które wykonują tylko jedną czynność. Są one podstawowymi składnikami programu, ponieważ każde działanie komputera jest przez nie wykonywane. W **Action!** są dwa rodzaje instrukcji prostych: instrukcje przypisania (w tym również wywołania funkcji) oraz wywołania procedur. Także dwa słowa kluczowe (EXIT i RETURN) są instrukcjami prostymi, lecz ze względu na specyficzne zastosowanie będą omówione później.

INSTRUKCJA PRZYPISANIA

Instrukcja przypisania służy do nadania wartości zmiennej. Jej najczęściej używanym formatem jest:

‹zmienna› = ‹wyrażenie arytmetyczne›

‹zmienna› może być podstawowego typu danych albo też może być tablicą, wskaźnikiem lub określeniem rekordu.

‹wyrażenie› MUSI być arytmetyczne. Jeżeli spróbujesz użyć wyrażenia logicznego, to wystąpi błąd, ponieważ kompilator **Action!** nie podstawia wartości liczbowej po obliczeniu wyrażenia logicznego.

Operatorem przypisania jest „=". Wskazuje on komputerowi, że chcesz przypisać nową wartość podanej zmiennej. Nie należy mylić go z logicznym „=". Pomimo że jest to ten sam znak, kompilator odczytuje go różnie, zależnie od kontekstu.

Poniższe przykłady ilustrują instrukcję przypisania. Zwróć uwagę na deklaracje zmiennych poprzedzające przykłady. Są one potrzebne, gdyż niektóre przykłady pokazują mieszanie typów, tzn. zmienna i przypisywana jej wartość są różnego typu.

```
BYTE b1,b2,b3,b4
INT i1
CARD c1
```

b3='D	umieszcza kod ATASCII znaku „D" w bajcie zarezerwowanym dla zmiennej b3.
b4=$44	umieszcza liczbę szesnastkową $44 w bajcie zarezerwowanym dla zmiennej b4 ($44 jest kodem ATASCII znaku „D", więc zmienne b3 i b4 zawierają teraz to samo).
b1=b4+16	dodaje 16 do wartości liczbowej zmiennej b4 i wynik umieszcza w bajcie zarezerwowanym dla zmiennej b1.
c1=23439-$07D8	umieszcza wartość 21431 ($53B7) w dwóch bajtach zarezerwowanych dla c1
i1=c1*(-1)	umieszcza wartość — 21431 ($AC49) w dwóch bajtach zarezerwowanych dla i1.
b2=i1	umieszcza wartość $49 (73) w bajcie zarezerwowanym dla b2. Zauważ, że komputer uwzględnia tylko LSB (MSB z i1 jest $AC, LSB — $49).
b2=b2+1	dodaje 1 do aktualnej wartości b2 i umieszcza sumę ponownie w b2. Zmienna b2 zawiera teraz $4A (74).

Zwróć uwagę na ostatni przykład. Jego format jest następujący:

‹zmienna› = ‹zmienna› ‹operator› ‹operand›

Ponieważ ten format instrukcji przypisania jest bardzo często używany przez programistów, to w **Action!** zastosowano do tego celu format skrócony:

‹zmienna› ==‹operator› ‹operand›

Operator musi być artmetyczny lub bitowy, a operand musi być wyrażeniem arytmetycznym. Oto kilka przykładów skróconego formatu:

b2==+1	jest równoważne	b2=b2+1
b2==-b1	jest równoważne	b2=b2-b1
b2==& $0F	jest równoważne	b2=b2 & $0F
b2==LSH (5+3)	jest równoważne	b2=b2 LSH (5+3)

Skrócony format zmniejsza ilość błędów popełnianych przy pisaniu programu, a także daje po skompilowaniu prostszy i krótszy kod wynikowy.

PROGRAMOWANIE

INSTRUKCJE STRUKTURALNE

Jeżeli dostępne byłyby tylko instrukcje proste, to czynności wykonywane przez komputer byłyby bardzo ograniczone:

Jedynym sposobem powtórzenia jakiejś grupy instrukcji byłoby powtórzenie ich w programie odpowiednią ilość razy w prawidłowej kolejności. Jeśli chciałbyś powtórzyć grupę dziesięciu instrukcji 10 razy, to musiałbyś wpisać 100 instrukcji.

Nie można by wykonywać grupy instrukcji warunkowo, to znaczy wykonywać je tylko wtedy, gdy spełniony jest jakiś warunek.

Celem instrukcji strukturalnych jest usunięcie tych i innych problemów. Instrukcje strukturalne dzielą się na dwie kategorie: instrukcje warunkowe i instrukcje pętli.

WYKONANIE WARUNKOWE

Wykonanie warunkowe pozwala na sprawdzenie wartości wyrażenia i wykonanie różnych instrukcji zależnie od rezultatu tego sprawdzenia. Ponieważ wyrażenie kontroluje warunkowe wykonanie, to nazywane jest ono wyrażeniem warunkowym.

Wykonanie warunkowe umożliwiają trzy instrukcje **Action!:** IF, WHILE, UNTIL. Dwie ostatnie są równocześnie instrukcjami pętli.

WYRAŻENIA WARUNKOWE

Ponieważ wyrażenie warunkowe jest wykorzystywane jako test, to może mieć tylko dwie wartości — prawda lub fałsz. Nie oznacza to jednak, że wyrażenie warunkowe jest jakimś nowym typem wyrażenia. W rzeczywistości wyrażenie warunkowe jest wyrażeniem logicznym lub arytmetycznym. Tylko interpretacja jest nieco inna. Poniższa tabelka pokazuje interpretację warunkową w zależności od typu wyrażenia:

typ wyrażenia	wynik normalny	wynik warunkowy
arytmetyczne	zero (0)	fałsz
	nie zero	prawda
logiczne	fałsz	fałsz
	prawda	prawda

INSTRUKCJA IF

Instrukcja IF (jeżeli) w **Action!** jest bardzo podobna do słowa „jeżeli" w języku polskim. Na przykład:

„Jeżeli mam 90 złotych lub więcej, to kupię hamburgera."

W **Action!** może to być wyrażone następująco:

```
BYTE pieniadze,
     hamburger=[90],
     paluszki=[80],
     hotdog=[60],
     ciastko=[20]
IF pieniadze>=90 THEN
   zakup(hamburger,pieniadze)
FI
```

UWAGA: zakup (hamburger, pieniądze) jest wywołaniem procedury i będzie dokładnie opisane później.

Z podanego przykładu widać, że podstawowym formatem instrukcji IF jest:

```
IF ‹wyrażenie warunkowe› THEN
    ‹instrukcja(e)›
FI
```

FI jest odwrotnie napisanym IF i jest słowem kluczowym, które wskazuje kompilatorowi koniec instrukcji IF. Ponieważ IF może dotyczyć większej liczby instrukcji, to konieczne jest zakończenie tych instrukcji przez FI. Bez tego słowa kompilator nie będzie wiedział, ile następujących po THEN instrukcji należy do struktury IF.

Oprócz podanego formatu podstawowego instrukcja IF ma jeszcze dwa warianty: ELSE i ELSEIF. Po polsku można to wyrazić następująco:

„Jeżeli mam 90 złotych lub więcej, to kupię hamburgera, w przeciwnym razie kupię paluszki."

Odpowiednikiem tego zdania w **Action!** jest:

```
IF pieniadze>=90 THEN
   zakup(hamburger,pieniadze)
ELSE
   zakup(paluszki,pieniadze)
FI
```

ELSEIF jest nieco odmienne:

„Jeżeli mam 90 złotych lub więcej, to kupię hamburgera, w przeciwnym razie jeżeli mam mniej niż 90 a i więcej niż 80, to kupię paluszki, w przeciwnym razie jeżeli mam mniej niż 80 i więcej niż 60, to kupię hotdoga, w przeciwnym razie kupię ciastko."

I zapis w **Action!**:

```
IF pieniadze>=90 THEN
   zakup(hamburger,pieniadze)
ELSEIF pieniadze>=80 THEN
   zakup(paluszki,pieniadze)
ELSEIF pieniadze>=60 THEN
   zakup(hotdog,pieniadze)
ELSE
   zakup(ciastko,pieniadze)
FI
```

Zauważ, że nie trzeba sprawdzać „pieniądze›=80 AND pieniądze‹90". Jest tak, ponieważ komputer wykonuje instrukcje kolejno z góry na dół. Jeżeli jeden przypadek jest prawdziwy, to odpowiednia grupa instrukcji jest wykonywana, a cała reszta instrukcji IF (razem ze wszystkimi następnymi ELSE i ELSEIF) jest pomijana. Tak więc gdy komputer doszedł do „pieniądze›=80", to wiadomo, że masz mniej niż 90 złotych, gdyż poprzednio musiał być sprawdzony warunek „pienądze›=90" i wynik był fałszem.

Wariant ELSEIF jest bardzo użyteczny, gdy chcesz kilkakrotnie sprawdzić zmienną, a dla różnych jej wartości są przewidziane różne instrukcje do wykonania.

INSTRUKCJA PUSTA

Instrukcja pusta nie robi nic. Po pokazaniu instrukcji wykonujących różne czynności, po wskazaniu konieczności, aby instrukcje coś wykonywały, teraz instrukcja, która nic nie robi? Są jednak pewne zastosowania dla takich instrukcji: pętle czasowe i przypadki ELSEIF.

Ponieważ nie opisywaliśmy jeszcze pętli, to powiedzmy po prostu, że pętle czasowe służą do opóźnienia wykonywania programu (np. gdy potrzebna jest przerwa między wyświetlaniem kolejnych linii na ekranie).

Użycie instrukcji pustej w przypadku ELSEIF zilustruje natomiast następujący przykład:

Scenariusz: piszesz program, który pozwala magazynierowi odszukać informacje o towarach przy użyciu ustalonych poleceń. Dostępne są polecenia: PRZYJĘCIE, WYDANIE, SZUKANIE i KONIEC, lecz polecenie SZUKANIE nie jest jeszcze zaimplementowane. Polecenie jest rozpoznawane po pierwszej literze, a więc należy porównać wprowadzony znak z P, W, S i K. Jednak SZUKANIE nie jest zrobione, co więc czynić, gdy naciśnięty został klawisz „S"? Po prostu nic, aż do czasu ukończenia operacji SZUKANIE. Odpowiedni fragment programu może wyglądać tak:

```
IF znak='P THEN
   zrobprzyjecie()
ELSEIF znak='W THEN
   zrobwydanie()
ELSEIF znak='S THEN
   ;**** tu jest instrukcja pusta
ELSEIF znak='K THEN
   zrobkoniec()
ELSE
   pomylka() ;**** zle polecenie
FI
```

Wszystkie „zrób..." są procedurami, które wykonują podane polecenia. Jeżeli spojrzysz na przypadek „znak= 'S", to zobaczysz, że nic tam nie jest robione. To jest właśnie instrukcja pusta. Gdy SZUKANIE jest gotowe, wystarczy tylko zamiast instrukcji pustej wstawić wywołanie procedury „zróbszukanie ()".

PĘTLE

Pętle są stosowane do powtarzania pewnych instrukcji. Jeżeli w jakimś celu chcesz wypełnić ekran gwiazdkami (*), to możesz wyświetlić każdą gwiazdkę oddzielną instrukcją lub możesz do tego użyć pętli. Wystarczy tylko wskazać, ile razy pętla ma wyświetlić pojedynczą gwiazdkę i to wszystko (oczywiście jeśli użyjesz prawidłowego formatu instrukcji).

Są dwa sposoby wskazania, ile razy pętla ma coś wykonać. Można podać konkretną liczbę lub użyć wyrażenia warunkowego i wykonywać pętlę zależnie od wartości tego wyrażenia. Pierwszy sposób jest stosowany w instrukcji FOR, a drugi w instrukcjach WHILE i UNTIL.

Co będzie, gdy nie wskażemy, ile razy ma być wykonana pętla? Co zrobić, gdy wyrażenie warunkowe nigdy nie osiągnie wartości kończącej pętlę? Taki przypadek jest zwany „pętlą bez końca". Jest tylko jeden sposób przerwania pętli bez końca — naciśnięcie klawisza ‹RESET›.

Action! udostępnia pętle w następujący sposób: Podstawowa pętla, gdy jest użyta samodzielnie, to nie ma końca. Oprócz tego są instrukcje kontrolujące pętle (FOR, WHILE, UNTIL), które ograniczają liczbę wykonań pętli. Zrobimy w ten sam sposób: najpierw poznamy strukturę pętli podstawowej, a następnie instrukcje kontrolujące pętle.

INSTRUKCJE DO i OD

„DO" i „OD" są używane do oznaczania odpowiednio początku i końca pętli podstawowej. Wszystko między nimi jest częścią tej pętli. Jak wcześniej powiedziano, sama pętla (tzn. bez żadnej instrukcji kontroli pętli) jest pętlą bez końca. Poniższy przykład ilustruje pętlę DO—OD. Nie przejmuj się instrukcjami „PROC" i „RETURN". Są one tu dołączone, aby program mógł być prawidłowo skompilowany i uruchomiony, a opisane będą później.

```
PROC razydwa()

   CARD i=[0],j

   DO             ;poczatek petli DO-OD
   i==+1          ;dodanie 1 do 'i'
   j=i*2          ;j rowne i*2
   PrintC(i)      ;zob. nizej
   Print(" razy 2 jest rowne ")
   PrintCE(j)
   OD             ;koniec petli DO-OD
RETURN
```

UWAGA: Słowa pisane dużymi i małymi literami (PrintC, Print, PrintCE), które znajdują się w przykładzie są procedurami bibliotecznymi **Action!.** Muszą one być użyte dla lepszego pokazania przykładu, a opis biblioteki **Action!** znajdzie się na końcu naszego kursu.

Po skompilowaniu i uruchomieniu programu zobaczysz coś takiego:

```
1 razy 2 jest rowne 2
2 razy 2 jest rowne 4
3 razy 2 jest rowne 6
4 razy 2 jest rowne 8
5 razy 2 jest rowne 10
6 razy 2 jest rowne 12
7 razy 2 jest rowne 14
8 razy 2 jest rowne 16
          ...
          ...
```

Punkty na końcu wskazują, że wyświetlanie będzie trwało dalej, aż do naciśnięcia ‹RESET›. Sama w sobie pętla DO—OD może być mniej lub bardziej użyteczna, lecz dopiero w połączeniu z instrukcjami kontrolującymi FOR, WHILE i UNTIL staje się jedną z najbardziej pożytecznych instrukcji.

UWAGA: naciśnięcie klawisza ‹BREAK› również przerywa ten program, ponieważ pętla używa operacji wejścia/wyjścia, a ‹BREAK› działa tylko wtedy (więcej informacji znajdziesz w opisie kompilatora).

Zawsze gdy w opisie formatu instrukcji kontroli pętli zobaczysz „‹pętla DO—OD›", pamiętaj, że oznacza to całą pętlę złożoną z różnych instrukcji ograniczonych instrukcjami DO i OD.

INSTRUKCJA EXIT

Instrukcja EXIT służy do opuszczenia dowolnej pętli. Powoduje ona przejście wykonywania programu do instrukcji następującej po „OD". Oto przykład

```
PROC razydwa()

   CARD i=[0],j

   DO             ;poczatek petli DO-OD
   i==+1          ;dodanie 1 do 'i'
   j=i*2          ;j rowne i*2
   PrintC(i)
   Print(" razy 2 jest rowne ")
   EXIT           ;tu jest wyjscie
   PrintCE(j)
   OD             ;koniec petli DO-OD
   ;*** stad jest kontynuowane
   ;*** wykonywanie po 'EXIT'
   PrintE("Koniec tablicy")
RETURN
```

i jego rezultat:

```
1 razy 2 jest rowne Koniec tablicy
```

Jak widać, instrukcja „PrintCE (j)" nigdy nie będzie wykonana. Instrukcja EXIT wymusza skok do instrukcji „PrintE („Koniec tablicy")". EXIT jest rzadko używana samodzielnie (jak wyżej), lecz staje się bardzo wygodna w połączeniu z instrukcją IF (daje np. warunkowe wyjście z pętli). Taki sposób jej wykorzystania pokazuje następny przykład.

```
PROC razydwa()

   CARD i=[0],j

   DO             ;poczatek petli DO-OD
   IF i=15 THEN
      EXIT        ;EXIT w IF
   FI
   i==+1
   j=i*2
   PrintC(i)
   Print(" razy 2 jest rowne ")
   PrintCE(j)
   OD             ;koniec petli DO-OD
   ;*** stad jest kontynuowane
   ;*** wykonywanie, gdy i=15
   PrintE("Koniec tablicy")
RETURN
```

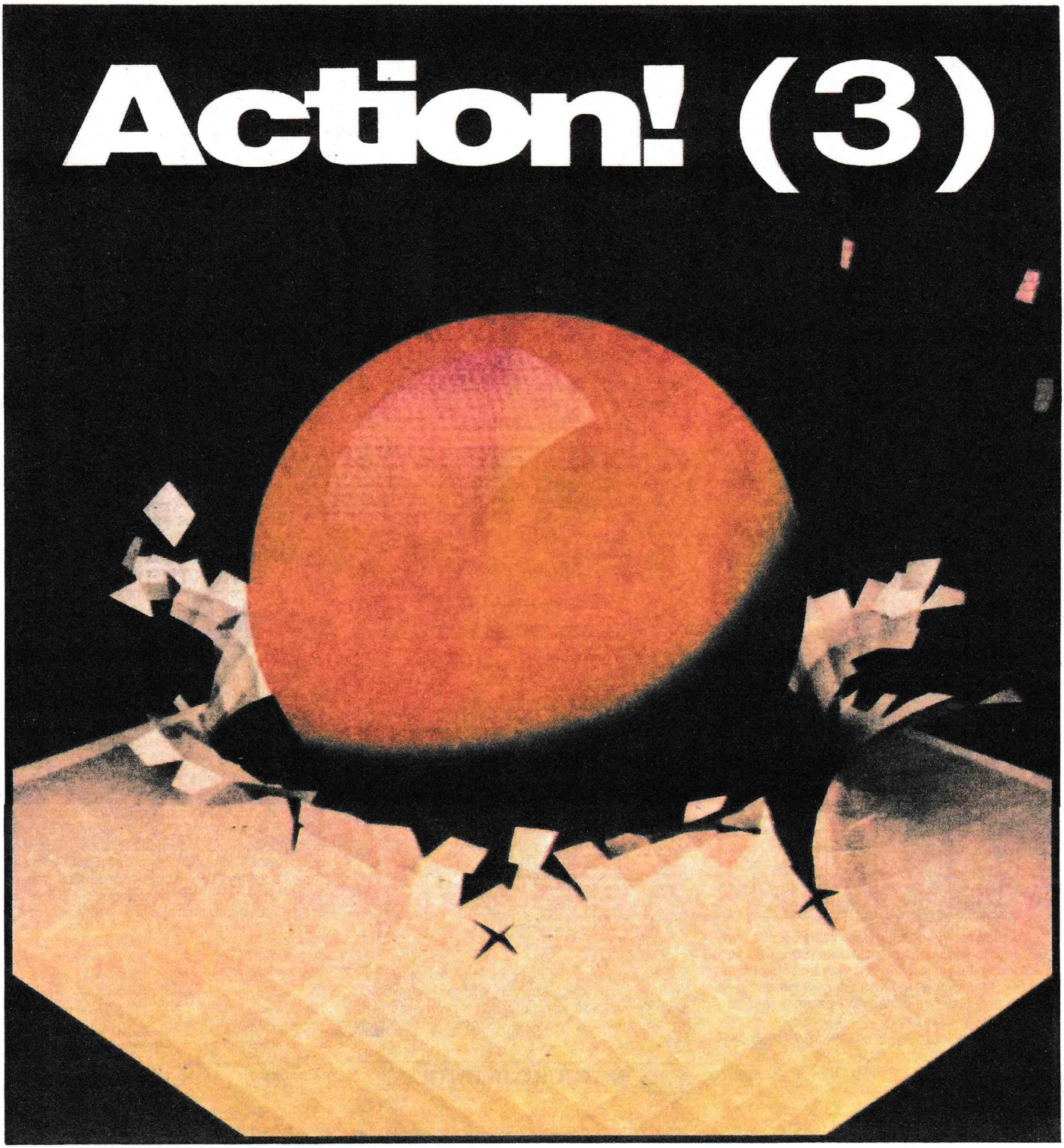
Wynik:

```
1 razy 2 jest rowne 2
2 razy 2 jest rowne 4
3 razy 2 jest rowne 6
4 razy 2 jest rowne 8
5 razy 2 jest rowne 10
6 razy 2 jest rowne 12
7 razy 2 jest rowne 14
8 razy 2 jest rowne 16
9 razy 2 jest rowne 18
10 razy 2 jest rowne 20
11 razy 2 jest rowne 22
12 razy 2 jest rowne 24
13 razy 2 jest rowne 26
14 razy 2 jest rowne 28
15 razy 2 jest rowne 30
Koniec tablicy
```

Zamienia to pętlę bez końca w pętlę z wyjściem. EXIT może kontrolować wykonywanie pętli, lecz nie oznacza to strukturalnej instrukcji kontroli pętli, gdyż działa ona tak tylko w połączeniu z IF.

Wojciech Zientara

Action! (3)



KONTROLA PĘTLI

Action! ma trzy instrukcje strukturalne, które kontrolują podstawową pętlę DO—OD: FOR, WHILE i UNTIL. Sformułowanie „kontrolują pętlę DO—OD" oznacza, że ograniczają one ilość powtórzeń pętli bez końca, przez co uzyskujemy pętlę z wyjściem do dalszej części programu. Kontrolowane pętle są jedną z najbardziej użytecznych i najczęściej używanych konstrukcji programowych.

Teraz zajmiemy się szerzej każdą instrukcją kontroli pętli kolejno, a następnie omówimy specjalną cechę wszystkich instrukcji strukturalnych: zagnieżdżanie.

INSTRUKCJA FOR

Instrukcja FOR służy do powtarzania pętli określoną liczbę razy. Wymaga ona własnej specjalnej zmiennej, zwanej licznikiem lub — bardziej naukowo — zmienną sterującą. We wszystkich przykładach zmienna ta ma nazwę „licz", aby była łatwa do odróżnienia, lecz możesz ją nazwać dowolnie. Format instrukcji FOR jest następujący:

FOR ‹licznik›=‹wart_pocz› TO ‹wart_końc› [STEP ‹krok›] ‹pętla DO—OD›

gdzie ‹licznik›	jest zmienną służącą do zliczania wykonań pętli
‹wart_pocz›	jest początkową wartością licznika
‹wart_końc›	jest końcową wartością licznika
‹krok›	jest liczbą, o którą komputer zwiększa licznik po każdym wykonaniu pętli (można to opuścić)
‹pętla DO—OD›	jest pętlą DO—OD bez końca

Zamiast objaśniania przy użyciu opisów i przenośni pokażemy kilka przykładów ilustrujących działanie instrukcji. Po każdym z nich następuje wydruk uzyskanego na ekranie wyniku.

Przykład 1:

```
PROC ahoj()

   BYTE licz   ;licznik dla petli FOR

   FOR licz=1 TO 5 ; nie ma STEP wiec
      DO           ; krok = 1
      PrintE("Ahoj przygodo!")
      OD
RETURN
```

Wynik 1:

```
Ahoj przygodo
Ahoj przygodo
Ahoj przygodo
Ahoj przygodo
Ahoj przygodo
```

Przykład 2:

```
PROC parzyste()

   BYTE licz   ;licznik dla petli FOR

   FOR licz=0 TO 16 STEP 2
      DO           ; krok = 2
      PrintB(licz)
      Print(" ")
      OD
RETURN
```

Wynik 2:

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

Spójrz ponownie na format instrukcji FOR. Zwróć uwagę, że nie mówiliśmy nic o użyciu zmiennych arytmetycznych jako ‹wart_pocz›, ‹wart_koń› lub ‹krok›. Jest to dozwolone i umożliwia uzyskanie pętli o zmiennej liczbie wykonań.

Jeżeli w środku pętli zmienisz wartości zmiennych użytych jako ‹wart_pocz›, ‹wart_koń› lub ‹krok›, to liczba wykonań pętli nie zmieni się, ponieważ po wejściu do pętli są one zapamiętywane jako wartości stałe. Jeśli są to zmienne, to przyjmowana jest ich wartość w chwili pierwszego wejścia do pętli.

Jeżeli wewnątrz pętli zmienisz wartość licznika, to możesz zmienić ilość pętli, ponieważ ‹licznik› jest w pętli zmienną. Jest tak, gdyż instrukcja FOR sama musi zmieniać wartość licznika po każdym wykonaniu pętli (FOR zwiększa ‹licznik› o wartość ‹krok›. Zmiany ‹wart_pocz›, ‹wart_koń› i ‹krok› wewnątrz pętli ilustruje następny przykład:

Przykład 3:

```
PROC zmianapetli()

  BYTE licz,
       poczatek=[1],
       koniec=[50]

  FOR licz=poczatek TO koniec
     DO
     poczatek=100 ;nie zmienia petli
     koniec=10    ;nie zmienia petli
     PrintBE(licz)
     licz==*2     ;ZMIENIA petle
     OD
RETURN
```

Wynik 3:

```
1
3
7
15
31
```

Zamieszczona niżej tabelka pokazuje, co dzieje się w pętli przy każdym jej wykonaniu. "wyk" pokazuje, które wykonanie pętli zostało zakończone, "zw licz" — wynik zwiększenia licznika przez FOR, "druk" — co jest wyświetlane na ekranie, a "licz==*2" pokazuje, jak instrukcja przypisania zmienia wartość licznika.

wyk	zw licz	druk	licz==*2
1	—	1	2
2	3	3	6
3	7	7	14
4	15	15	30
5	31	31	62

Po piątym wykonaniu pętli licznik jest równy 62. Jest to więcej niż ‹wart_koń› (50), więc pętla kończy się po jedynie 5 wykonaniach zamiast po 50 . Manipulowanie licznikiem wewnątrz pętli może dać ciekawe rezultaty i czasem jest bardzo użyteczne.

A oto przykład wspominanej wcześniej pętli czasowej:

```
BYTE licz

FOR licz=1 TO 250
   DO
   ;*** tu jest instrukcja pusta
   OD
```

Służy ona jedynie do opóźnienia wykonywania programu. Stosowana jest często w grach i innych programach, które wymagają dokładnego regulowania czasu pracy.

UWAGA: Jeżeli napiszesz pętlę FOR, w której licznik przekracza limit typu danej (255, gdy licznik jest typu BYTE i 65535, gdy CARD), to pętla nie będzie miała końca, ponieważ wtedy licznik nie może zostać zwiększony do wartości przekraczającej ‹wart_koń›.

INSTRUKCJA WHILE

Instrukcja WHILE (a także UNTIL) jest stosowana, gdy nie chcesz, aby pętla była wykonywana określoną wcześniej ilość razy. WHILE służy do wykonywania pętli tak długo, jak podane wyrażenie warunkowe jest prawdziwe. Ma ona format:

WHILE ‹wyr_war›
‹pętla DO—OD›

gdzie ‹wyr_war› jest wyrażeniem warunkowym kontrolującym pętlę
‹pętla DO—OD› jest pętlą DO—OD bez końca

Ponieważ wyrażenie warunkowe jest obliczane na początku pętli, to ‹pętla DO—OD› może wcale nie być wykonywana. Taki przypadek nie występuje w instrukcji UNTIL. Kolejne przykłady wykorzystują instrukcję WHILE.

Przykład 1:

```
PROC silnia()
;*** procedura podaje silnie, az do
;okreslonej wartosci (zmienna limit)

   CARD silnia=[1],  ;silnia 'num'
        numer=[1],   ;licznik
        limit=[6000] ;gorna granica

   Print("Silnie mniejsze niz ")
   PrintC(limit)
   PrintE(":")
   PutE()
   WHILE silnia*numer<limit
      DO        ;poczatek petli WHILE
      silnia==*numer
      PrintC(numer)
      Print("! silnia jest rowne ")
      PrintCE(silnia)
      numer==+1 ;zwiekszenie licznika
      OD        ;koniec petli WHILE
RETURN          ;koniec procedury
```

Wynik 1:

```
Silnie mniejsze niz 6000:

1 silnia jest rowne 1
2 silnia jest rowne 2
3 silnia jest rowne 6
4 silnia jest rowne 24
5 silnia jest rowne 120
6 silnia jest rowne 720
7 silnia jest rowne 5040
```

Przykład 2:

```
PROC numerkiwhile()
;*** procedura gry w numerki
; uzywajaca petli WHILE

   BYTE numer,          ;zgadywana liczba
        liczba=[200]    ;odpowiedz

   PrintE("Zapraszamy do gry w numerki.")
   PrintE("Pomyslalem liczbe od 0 do 100.")
   numer=Rand(101)
   WHILE liczba<>numer
      DO        ;poczatek petli WHILE
      Print("Jaka to liczba? ")
      liczba=InputB()
      IF liczba<numer THEN
         PrintE("Za malo, jeszcze raz")
      ELSEIF liczba>numer THEN
         PrintE("Za duzo, jeszcze raz")
      ELSE
         PrintE("Gratulacje!!!")
         PrintE("Odgadles")
      FI        ;koniec sprawdzania liczby
      OD        ;koniec petli WHILE
RETURN          ;koniec procedury
```

Wynik 2:

```
Zapraszamy do gry w numerki.
Pomyslalem liczbe od 0 do 100.
Jaka to liczba? 50
Za malo, jeszcze raz
Jaka to liczba? 60
Za duzo, jeszcze raz
Jaka to liczba? 55
Za malo, jeszcze raz
Jaka to liczba? 57
Gratulacje!!!
Odgadles
```

Zwróć uwagę, jak duże możliwości daje użycie instrukcji warunkowych (np. IF) wewnątrz pętli. Pozwala to komputerowi wykonywać różne warianty instrukcji w różnych przejściach pętli.

INSTRUKCJA UNTIL

W poprzednim ustępie powiedzieliśmy, że pętla WHILE może nie zostać wykonana ani razu, ponieważ wyrażenie warunkowe jest sprawdzane naa początku pętli. Format instrukcji UNTIL jest taki, że pętla musi być wykonana przynajmniej jeden raz. Gdy zobaczysz ten format, zrozumiesz dlaczego tak jest:

DO
‹instrukcja›
UNTIL ‹wyr_war›
OD

Wygląda to jak zwykła pętla DO—OD, aż do miejsca tuż przed „OD". Umieszczone tu UNTIL kontroluje wyjście z pętli bez końca poprzez wyrażenie warunkowe. Jeżeli ‹wyr_war› jest prawdą, to wykonywanie programu jest kontynuowane od instrukcji następującej po OD, w przeciwnym razie wykonywane jest następne przejście pętli. Pamiętaj, że UNTIL musi być umieszczone bezpośrednio przed OD. Oto przykład:

```
PROC numerkiuntil()
;*** procedura gry w numerki
; uzywajaca petli UNTIL

   BYTE numer,          ;zgadywana liczba
        liczba          ;odpowiedz

   PrintE("Zapraszamy do gry w numerki.")
   PrintE("Pomyslalem liczbe od 0 do 100.")
   numer=Rand(101)
      DO        ;poczatek petli UNTIL
      Print("Jaka to liczba? ")
      liczba=InputB()
      IF liczba<numer THEN
         PrintE("Za malo, jeszcze raz")
      ELSEIF liczba>numer THEN
         PrintE("Za duzo, jeszcze raz")
      ELSE
         PrintE("Gratulacje!!!")
         PrintE("Odgadles")
      FI        ;koniec sprawdzania liczby
      UNTIL liczba=numer   ;kontrola petli
      OD        ;koniec petli UNTIL
RETURN          ;koniec procedury
```

Wynik:

```
Zapraszamy do gry w numerki.
Pomyslalem liczbe od 0 do 100.
Jaka to liczba? 50
Za malo, jeszcze raz
Jaka to liczba? 60
Za duzo, jeszcze raz
Jaka to liczba? 55
Za malo, jeszcze raz
Jaka to liczba? 57
Gratulacje!!!
Odgadles
```

Jest to ten sam przykład, co w opisie WHILE, lecz teraz wykorzystuje pętlę UNTIL. Zwróć uwagę, że nie trzeba nadawać wartości początkowej zmiennej „liczba". Spowodowane jest to sprawdzaniem warunku „numer=liczba" dopiero po podaniu liczby przez użytkownika. Jest to jedna z zalet pętli UNTIL, a wynika ze sprawdzania warunku dopiero na końcu pętli. WHILE sprawdza warunek na początku pętli i wymaga, aby „liczba" miała już wartość.

ZAGNIEŻDŻANIE INSTRUKCJI STRUKTURALNYCH

Jak powiedzieliśmy wcześniej, instrukcje strukturalne składają się z innych instrukcji. Instrukcjami wewnątrz instrukcji strukturalnych mogą być zarówno instrukcje proste, jak i inne instrukcje strukturalne. Umieszczenie instrukcji strukturalnej wewnątrz innej instrukcji strukturalnej nazywa się zagnieżdżeniem.

W opisach instrukcji WHILE i UNTIL możesz zobaczyć przykłady zagnieżdżenia instrukcji IF w pętlach WHILE i UNTIL. Taki rodzaj zagnieżdżenia jest oczywisty i nie będzie szerzej omawiany. Zajmiemy się teraz wielokrotnym zagnieżdżeniem instrukcji strukturalnych tego samego typu (IF wewnątrz IF, FOR w FOR itd.).

Zagnieżdżenie instrukcji IF może spowodować kłopot z określeniem, które ELSE jest związane z jakimś IF. Kompilator unika pomyłek dzięki łączeniu w pary instrukcji IF i FI. FI jest łączone z ostatnią instrukcją IF, która jeszcze nie ma pary. Na przykład:

```
+ IF <wyr_A> THEN
|   + IF <wyr_B> THEN
|   |     <instrukcje>
|   | ELSEIF <wyr_C> THEN ;* do IF <wyr_B
|   |   + IF <wyr_D> THEN
|   |   |     <instrukcje>
|   |   | ELSE              ;* do IF <wyr_D>
|   |   |     <instrukcje>
|   |   + FI                ;* koniec IF <wyr_D>
|   + FI                    ;* koniec IF <wyr_B>
| ELSEIF <wyr_E> THEN       ;* do IF <wyr_A>
|     <instrukcje>
| ELSE                      ;* do IF <wyr_A>
|     <instrukcje>
+ FI                        ;* koniec IF <wyr_A>
```

Linie wskazują pary instrukcji IF—FI, a komentarze pokazują, do której instrukcji IF należą poszczególne FI i ELSEIF. Orientację ułatwiają dodatkowo wcięcia wydruku.

Następny przykładowy program zawiera zagnieżdżone pętle FOR.

```
PROC tabliczkamnozenia()
;*** procedura podaje tabliczke
;mnozenia w zakresie do 10*10

BYTE l1,  ;licznik zewnetrznej petli
     l2   ;licznik wewnetrznej petli

  FOR l1=1 TO 10 ;petla zewnetrzna
     DO
     IF l1<10 THEN
        Print(" ")
     FI
     PrintB(l1)
     FOR l2=1 TO 10 ;petla wewnetrzna
        DO
        IF l1*l2<10 THEN
           Print("  ")
        ELSEIF l1*l2<100 THEN
           Print(" ")
        ELSE
           Print(" ")
        FI
        PrintB(l1*l2)
        OD      ;koniec petli wewnetrznej
     PutE()
     OD         ;koniec petli zewnetrznej
RETURN          ;koniec procedury
```

Wynik:

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

Jak możesz zobaczyć w powyższych przykładach, zagnieżdżenie może być bardzo pożyteczne, jeśli wiesz, co chcesz uzyskać.

PROCEDURY I FUNKCJE

Procedury i funkcje czynią program w **Action!** bardziej czytelnym i użytecznym. Prawie wszystko co robimy jest w taki czy inny sposób procedurą lub funkcją. Proszę zobaczyć:

PROCEDURY	FUNKCJE
Mycie naczyń	Odczyt z notesu
Gotowanie obiadu	Liczenie pieniędzy
Przejazd do pracy	

Co robią te procedury i funkcje? Każda z nich:
1. jest grupą powiązanych czynności wykonywanych dla zrealizowania określonego zadania;
2. ma ustaloną kolejność, w której te czynności są wykonywane.

W języku komputerowym wygląda to tak samo. Układasz grupę czynności, które wykonują pojedynczą, większą operację w procedurę lub funkcję i nadajesz jej nazwę. Gdy chcesz wykonać tą operację wystarczy użyć nazwy procedury lub funkcji. Jest to nazywane wywołaniem procedury lub funkcji. Oczywiście procedura lub funkcja musi być już zdefiniowana.

A jaka jest różnica między procedurą i funkcją? Obie są uporządkowanym zbiorem czynności wykonującym pewną operację, po co więc dwie nazwy dla takiej samej konstrukcji? Ponieważ nie są to takie same konstrukcje. Funkcja ma dodatkową własność: wykonuje pewną operację i zwraca (przekazuje do miejsca, z którego była wywołana) wartość.

W tabelce umieszczonej na początku tej części jako przykład funkcji został podany „Odczyt z notesu". Dlaczego? Ponieważ w celu odczytania z notesu trzeba wykonać kilka czynności, a na koniec jako wynik otrzymujemy numer telefonu. Ten numer jest zwracany i może być użyty w innej operacji (np. aby umówić się na randkę).

UWAGA: często stosuje się określenie „procedura" dla oznaczenia procedury lub funkcji. Jest to spowodowane niedoskonałością tłumaczenia z języka angielskiego, gdzie „procedure" i „function" oznaczają to samo co w języku polskim, lecz słowo „routine" oznaczające odrębną część programu (procedurę lub funkcję) nie ma specjalnego odpowiednika (czasem mówi się podprogram, ale to nie to samo).

PROCEDURY

Procedury łączą grupę instrukcji w blok z nazwą, który może być wywołany przez podanie nazwy. Aby wykorzystać procedury w **Action!** musisz nauczyć się dwóch rzeczy: deklarowania i wywoływania procedur. Trzy następne ustępy pokażą, jak to zrobić i podadzą kilka przykładów wykorzystania procedur w **Action!**

DEKLARACJA PROC

Słowo kluczowe „PROC" jest stosowane w **Action!** do wskazania początku deklaracji procedury. Konstrukcja procedury składa się z grupy instrukcji, które mają na początku nazwę i inne informacje, a na końcu instrukcję RETURN. A oto format tej konstrukcji:

```
PROC ‹nazwa› [=‹adres›] ([‹lista parametrów›])
     [‹deklaracje zmiennych›]
     [‹lista instrukcji›]
  RETURN
```

gdzie

PROC	jest słowem kluczowym oznaczającym deklarację procedury
‹nazwa›	jest nazwą procedury
‹adres›	określa adres początkowy procedury
‹lista parametrów›	jest wykazem parametrów wymaganych przez procedurę
‹deklaracje zmiennych›	jest to lista zmiennych deklarowanych lokalnie w tej procedurze (nie istniejących poza nią)
‹lista instrukcji›	są to instrukcje zawarte w procedurze
RETURN	oznacza koniec procedury

UWAGA! ‹lista parametrów›, ‹deklaracje zmiennych› i ‹lista instrukcji› mogą być opuszczone. Nigdy zapewne tego nie użyjesz, ale prawidłowa deklaracja procedury może wyglądać i tak:

```
PROC nic()      ;nawiasy są konieczne!
RETURN
```

Nie robi ona nic, lecz wpisanie takiej „pustej" procedury może być użyteczne przy pisaniu dużych programów. Służy to do zastępowania nie napisanych jeszcze procedur przy próbnym uruchamianiu niedokończonego programu.
RETURN i ‹lista parametrów› zostaną opisane dalej, a teraz przykład pokazujący pozostałe elementy deklaracji:

```
PROC numerkiuntil()
;*** procedura gry w numerki
; uzywajaca petli UNTIL

  BYTE numer,          ;zgadywana liczba
       liczba          ;odpowiedz

  PrintE("Zapraszamy do gry w numerki.")
  PrintE("Pomyslalem liczbe od 0 do 100.")
  numer=Rand(101)
    DO                 ;poczatek petli UNTIL
    Print("Jaka to liczba?  ")
    liczba=InputB()
    IF liczba<numer THEN
      PrintE("Za malo, jeszcze raz")
    ELSEIF liczba>numer THEN
      PrintE("Za duzo, jeszcze raz")
    ELSE
      PrintE("Gratulacje!!!")
      PrintE("Odgadles")
    FI                 ;koniec sprawdzania liczby
    UNTIL liczba=numer ;kontrola petli
    OD                 ;koniec petli UNTIL
RETURN                 ;koniec procedury
```

Jest to ten sam program, który był przykładem przy opisie instrukcji UNTIL, lecz wtedy nie rozumiałeś jeszcze instrukcji PROC i znajdujących się po niej deklaracji zmiennych. Już na początku pisaliśmy, że program w **Action!** wymaga deklaracji procedury lub funkcji, aby mógł być skompilowany. Powyższy przykład deklaracji procedury, jest więc równocześnie prawidłowym programem w **Action!** i oczywiście może być skompilowany i uruchomiony. Wynik będzie taki sam jak podany w opisie UNTIL:

```
Zapraszamy do gry w numerki.
Pomyslalem liczbe od 0 do 100.
Jaka to liczba?  50
Za malo, jeszcze raz
Jaka to liczba?  60
Za duzo, jeszcze raz
Jaka to liczba?  55
Za malo, jeszcze raz
Jaka to liczba?  57
Gratulacje!!!
Odgadles
```

Ostatnią instrukcją w przykładzie jest RETURN. Zobaczymy więc, do czego ona służy.

INSTRUKCJA RETURN

RETURN nakazuje kompilatorowi opuszczenie procedury i powrót do miejsca jej wywołania. Jeśli Twój program wywołał procedurę, to jego wykonywanie będzie kontynuowane od instrukcji następującej po wywołaniu procedury. Jeżeli kompilowałeś pojedynczą procedurę (lub program złożony z jednej procedury), to sterowanie zostanie przekazane do monitora **Action!**
UWAGA: kompilator może nie wykryć braku RETURN na końcu procedury. W takim przypadku najczęściej następuje zawieszenie komputera. Dotyczy to także funkcji.

Można użyć więcej niż jedno RETURN w procedurze. Na przykład, jeśli procedura zawiera instrukcję IF z kilkoma wariantami ELSEIF, to możesz żądać opuszczenia procedury po jednym lub po kilku przypadkach ELSEIF. Oto przykład ilustrujący tą możliwość:

```
PROC testpolecenia()
;*** Ta procedura sprawdza, czy
;podane polecenie jest prawidlowe.
;Prawidlowe sa polecenia 0,1,2 i 3.
;Zle polecenie powoduje meldunek
;bledu i powrot do miejsca
;wywolania procedury.

  BYTE pol

  Print("Polecenie>> ")
  pol=InputB()
  IF pol>3 THEN
    PrintE("Bledne polecenie")
    RETURN
  ELSEIF pol=0 THEN
    ‹instrukcja 0›
  ELSEIF pol=1 THEN
    ‹instrukcja 1›
  ELSEIF pol=2 THEN
    ‹instrukcja 2›
  ELSEIF pol=3 THEN
    ‹instrukcja 3›
  FI
RETURN
```

Zwróć uwagę na RETURN po pierwszym warunku, który sprawdza prawidłowość polecenia. Możesz nie chcieć, aby były sprawdzane pozostałe przypadki, gdy polecenie jest nieprawidłowe, więc następuje tylko wyświetlenie meldunku o błędzie i opuszczenie procedury przez RETURN.

WYWOŁYWANIE PROCEDURY

Wielokrotnie widziałeś już wywołanie procedury, lecz prawdopodobnie nie wiedziałeś, że to właśnie to. We wcześniejszych przykładach używaliśmy już procedur bibliotecznych i właśnie stosowaliśmy ich wywołanie. Jego format jest bardzo prosty:

‹nazwa› ([‹lista parametrów›])

gdzie ‹nazwa›	jest nazwą procedury, którą chcesz wywołać
‹lista parametrów›	zawiera wartości, która chcesz przekazać do procedury jako parametry

Oto przykład:

```
PrintE("Witamy w krainie Bajtka")

silnia()

numerki()

BYTE z
CARD a
koniec(a,z)
```

Oczywiście musisz zadeklarować procedury „silnia", „numerki" i „koniec" przed użyciem ich tutaj. „PrintE" jest procedurą biblioteczną, więc nie jest deklarowana przez Ciebie, lecz jest już zadeklarowana w bibliotece **Action!**. Zwróć uwagę, że nawiasy są wymagane, nawet jeśli procedura nie ma parametrów. Gdy wywoływana procedura ma parametry, w wywołaniu nie może być więcej parametrów niż w deklaracji procedury (lecz może być mniej). Dokładnie opiszemy to później.

FUNKCJE

Podstawową różnicą między procedurą i funkcją jest — jak już wiesz — to, że funkcja zwraca wartość. Powoduje to odmienny sposób jej deklarowania i wywoływania. Ponieważ funkcja zwraca wartość, to musi być użyta w miejscu, w którym użycie wartości jest prawidłkowe (np. w wyrażeniu arytmetycznym).

DEKLARACJA FUNC

Deklaracja funkcji jest bardzo podobna do deklaracji procedury poza tym, że trzeba wskazać,

jaki jest typ wartości zwracanej przez funkcję (BYTE, CARD lub INT) oraz która to wartość. Format deklaracji jest więc następujący:

‹typ› FUNC ‹nazwa› [=‹adres›] ([‹lista parametrów›])
[‹deklaracje zmennych›]
[‹lista instukcji›]
RETURN (‹wyrażenie arytm›)

gdzie

‹typ›	jest typem wartości zwracanej przez funkcję
FUNC	jest słowem kluczowym oznaczającym deklarację funkcji
‹nazwa›	jest nazwą funkcji
‹adres›	określa adres początkowy funkcji
‹lista parametrów›	jest wykazem parametrów wymaganych przez funkcję
‹deklaracje zmiennych›	jest to lista zmiennych deklarowanych lokalnie w tej funkcji (nie istniejących poza nią)
‹lista instrukcji›	są to instrukcje zawarte w funkcji
RETURN	oznacza koniec funkcji
‹wyrażenie arytm›	jest wartością, która zostanie zwrócona przez funkcję

Tak jak w deklaracji procedury ‹lista parametrów›, ‹deklaracje zmiennych› i ‹lista instrukcji› mogą być opuszczone. W przypadku procedury może to być użyteczne tylko w jednym przypadku. Natomiast w funkcji jest to stosowane znacznie częściej, jak w poniższym przykładzie:

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

Ta funkcja pobiera liczbę typu CARD i zwraca jej kwadrat. Mówiliśmy, że zwracana wartość jest w formie wyrażenia arytmetycznego. W naszym przykładzie jest to "(x*x)".

W kolejnym przykładzie zwracana wartość jest po prostu nazwą zmiennej.

```
BYTE FUNC polecenie()
;*** Ta funkcja odczytuje numer
;polecenia i sprawdza, czy jest
;mniejszy niz 8. W przeciwnym
;razie ponownie pyta o polecenie.

   BYTE polecenie,  ;numer polecenia
        blad        ;=1, gdy blad

   DO
   Print("POLECENIE>  ")
   polecenie=InputB()
   IF polecenie<1 OR polecenie>7 THEN
      blad=1
      Print("Zle polecenie! ")
      PrintE("dozwolone tylko 1-7.")
   ELSE
      blad=0
   FI
   UNTIL blad=0
   OD
RETURN (polecenie)
```

UWAGA! ‹wyrażenie arytm› w instrukcji RETURN musi być zawsze w nawiasach.

INSTRUKCJA RETURN

Zapewne zauważyłeś, że w formacie deklaracji funkcji RETURN jest użyte w inny sposób niż w deklaracji procedury. W funkcji następuje po nim (‹wyrażenie arytm›). Pozwala to funkcji zwracać wartość. Jeśli spróbujesz umieścić (‹wyrażenie arytm›) po RETURN w deklaracji procedury, otrzymasz błąd, gdyż procedura nie może zwracać wartości.

Pomimo pewnych różnic w użyciu RETURN, jest jedno ważne podobieństwo: zarówno w procedurze jek i w funkcji można kilkakrotnie użyć RETURN. Następny przykład pokazuje wielokrotne użycie RETURN w funkcji:

Funkcja „kwadrat" zadeklarowana poprzednio zwraca wartość kwadratu liczby typu CARD, lecz nie sprawdza, czy wystąpiło przepełnienie. Jeżeli podniesiesz do kwadratu 256, to otrzymasz 65536, a więc o jeden więcej niż zakres liczb CARD (65535). W rezulatacie funkcja zwróci wartość 65536−65535=1. Są dwa sposoby usunięcia tej wady:

1. Wymaganie, aby liczba podnoszona do kwadratu była typu BYTE, czyli czyni niemożliwym wprowadzenie liczby większej niż 255.

2. Sprawdzanie w samej funkcji, czyli wystąpi przepełnienie. Poniższy przykład ilustruje drugi sposób:

```
CARD FUNC kwadrat(CARD x)
;*** Ta funkcja sprawdza, czy 'x'
;nie spowoduje przepelnienia i
;jesli nie, to zwraca jego kwadrat.
;Jesli tak, to zwraca 0.

   IF x>255 THEN
      PrintE("Za duza liczba")
      RETURN (0)   ;zwraca zero
   FI
RETURN (x*x)   ;zwraca kwadrat x
```

Widzisz jakie to proste? Wielokrotne użycie RETURN jest szczególnie użyteczne przy sprawdzaniu wielu warunków, z których każdy wymaga zwrócenia innej wartości.

WYWOŁYWANIE FUNKCJI

Widziałeś już dwa przykłady wywołania funkcji. Możesz je znaleźć w przykładach do opisu WHILE i UNTIL. Jeśli spojrzysz do tych przykładów, zobaczysz następujące linie:

numer=Rand(101)
liczba=InputB()

Pierwsza jest przykładem funkcji wymagającej parametru, a druga — wywołania bez parametrów. Obie te funkcje są funkcjami bibliotecznymi. „Rand" zwraca wartość losową od zera do liczby podanej jako parametr zmniejszonej o jeden (0‹=numer‹=100).

„InputB" odczytuje wartość bajtu z urządzenia edytor. Oczywiście obie funkcje zwracają wartość. Ponieważ musi być ona użyta prawidłowo, to wywołania funkcji muszą być zastosowane w wyrażeniach arytmetycznych. W obu powyższych przykładach wyrażenia składają się tylko z wywołań funkcji i są użyte w instrukcjach przypisania.

Wywołanie funkcji może być użyte w dowolnym wyrażeniu arytmetycznym, z jednym wyjątkiem: **NIE WOLNO** użyć wywołania funkcji w wyrażeniu, które jest parametrem wywołania lub deklaracji procedury albo funkcji.

Przykład: x=kwadrat(2*Rand(50)) ;Źle

A teraz przykłady prawidłowych wywołań funkcji:

x=5*Rand(201)
c=kwadrat(x)−100/x
IF wsk‹›Peek($8000)
znak=duzalitera(znak)

„Peek" i „Rand" są funkcjami bibliotecznymi, więc nie muszą być deklarowane przez Ciebie, lecz „kwadrat" i „duzalitera" są funkcjami użytkowanika, więc trzeba jej zadeklarować przed wywołaniem.

UWAGA: można, choć nie jest to zalecane, wywołać funkcję tak, jakby to była procedura. Zwracana wartość jest wtedy ignorowana.

DZIEDZINA ZMIENNYCH

Termin „dziedzina zmiennej" jest stosowany do określenia zasięgu działania i ważności zmiennej. W Basicu zmienne, niezależnie od miejsca, w którym zostały określone, są dostępne w całym programie. Takie zmienne nazywamy zmiennymi globalnymi. W **Action!** oprócz zmiennych globalnych występują zmienne lokalne. Są to zmienne, które, są „widoczne" tylko w ograniczonym obszarze, np. w procedurze. Zmienna lokalna określona w jakiejś procedurze nie istnieje poza nią. Unika się dzięki temu przypadkowej zmiany wartości zmiennej, gdyż w różnych procedurach mogą być użyte zmienne lokalne o takich samych nazwach i nie mające ze sobą żadnego związku. Najłatwiej będzie to zrozumieć na konkretnym przykładzie.

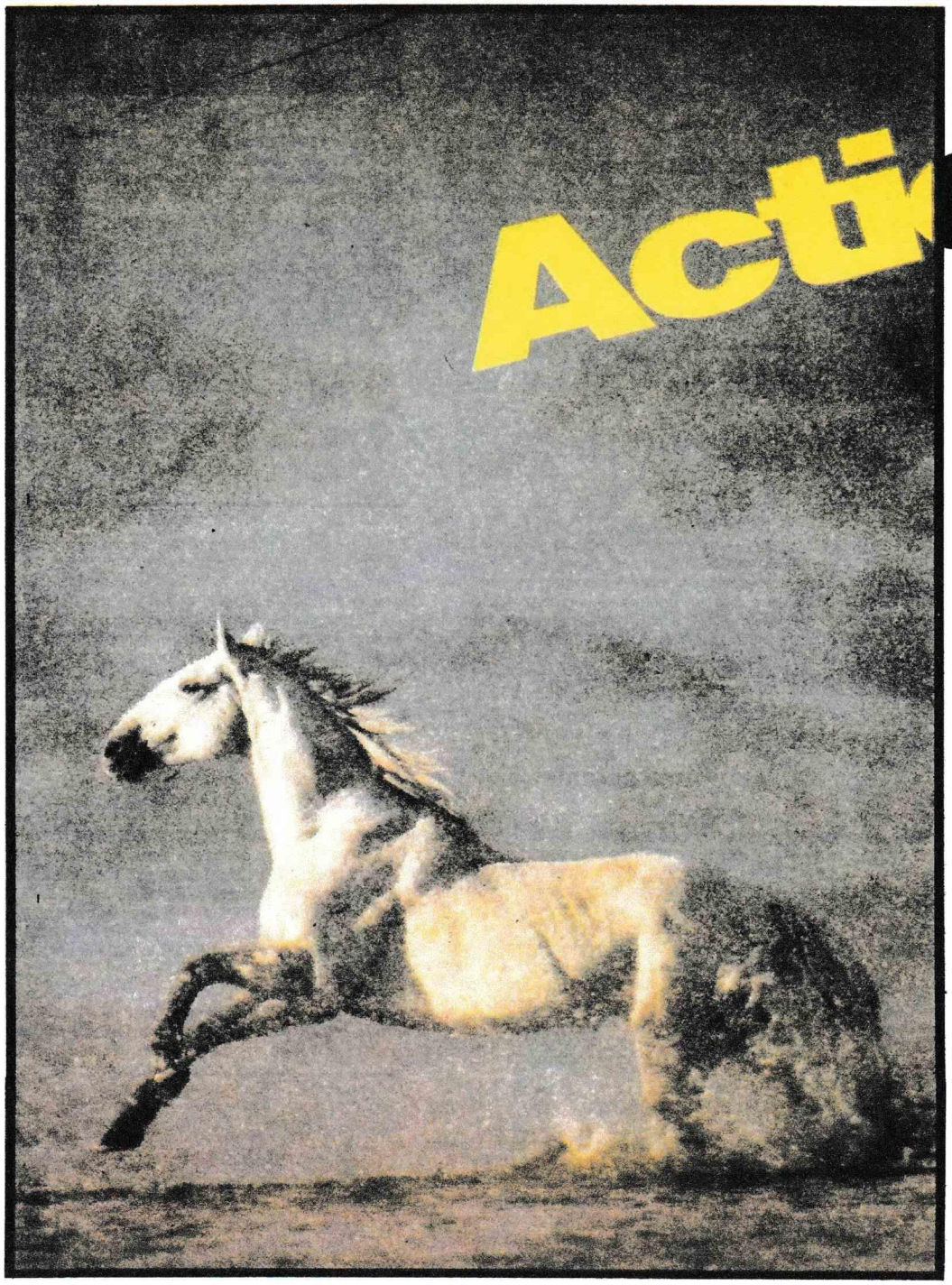
```
MODULE ;deklaracja zmiennych globalnych

CARD iloscgier=[0],
     iloscprob=[10],
     trafne=[0]

PROC wstep()
;**** Ta procedura wyswietla instrukcje
;do gry na ekranie.

   CARD licznik

   PrintE("Witaj w zgaduj-zgaduli.")
   PrintE("Pomysle liczbe od 0 do 100.")
   PrintE("Gdy zapytam Cie, jaka to liczba,")
   PrintE("musisz wpisac odpowiedz.")
   PutE()
   PrintE("Bede pamietal, ile gier rozegrales")
   PrintE("i powiem Ci, ile razy probowales")
   PrintE("odgadnac liczbe, ale najpierw")
   PrintE("musze znac dozwolona ilosc prob.")
   PutE()
   Print("  Wpisz tu ilosc prob —> ")
   iloscprob=InputC()
   FOR licznik=0 TO 2500  ;petla opozniajaca
      DO
      OD
   Put($7D)       ;czyszczenie ekranu
RETURN            ;koniec PROCedury wstep

PROC wynik()
;**** Ta procedura wyswietla aktualny stan gry.

   Print("Rozegrales juz ")
   PrintC(iloscgier)
   PrintE(" gier,")
   Print("i w ")
   PrintC(trafne)
   PrintE("grach odgadles liczbe")
   PrintE("zadajac mniej niz")
   PrintC(iloscprob)
   PrintE(" pytan.")
   PutE()
RETURN            ;koniec PROCedury wynik

PROC rozgrywka()

   CARD proba,   ;nume kolejnej proby
        licznik  ;licznik petli opozniajacej

   BYTE numer,   ;numer do odgadniecia
        liczba   ;odpowiedz gracza

   PrintE("Pomysle sobie liczbe...")
   FOR licznik=0 TO 4500   ;petla
      DO      ;symulujaca namyslanie sie
      OD      ;przez komputer
   PutE()
   PrintE("Juz, teraz Twoja kolei!")
   PutE()
   numer=Rand(101)   ;liczba do odgadniecia
   proba=0   ;ustawienie liczby prob na 0
   DO              ;poczatek petli UNTIL
      Print("Jaka jest Twoja odpowiedz?  ")
      liczba=InputB() ;zgadywanie liczby
      proba==+1  ;zwiekszenie numeru proby
      IF liczba<numer THEN      ;za malo
         PrintE("Za malo, jeszcze raz")
      ELSEIF liczba>numer THEN ;za duzo
         PrintE("Za duzo, jeszcze raz")
      ELSE      ;dobra odpowiedz
         PrintE("Moje gratulacje !!!")
         Print("Odgadles w ")
         PrintC(proba)
         PrintE(" probie.")
         IF proba<iloscprob THEN
            trafne==+1
         FI
      FI      ;koniec testowania liczby
   UNTIL liczba=numer   ;kontrola petli
   OD                   ;koniec petli
RETURN      ;koniec PROCedury rozgrywka

BYTE FUNC stop()
;**** Ta funkcja sprawdza, czy gracz chce
;grac dalej.

   BYTE znowu

   PrintE("Czy chcesz zagrac")
   Print("jeszcze raz? (T lub N)  ")
   znowu=GetD(1)  ; pobranie odpowiedzi z K:
   PutE()
   IF znowu='N OR znowu='n THEN
      RETURN(1)      ;nie chce
   FI
RETURN(0)            ;koniec FUNKCji stop

PROC glowna()

   Close(1)      ;na wszelki wypadek
   Open(1,"K:",4,0) ;otwarcie K: do odczytu
   wstep()       ;wyswietlenie instrukcji
   DO
      iloscgier==+1  ;kolejny numer gry
      rozgrywka()    ;zgadywanie
      wynik()        ;wyswietlenie wyniku
   UNTIL stop()      ;gdy nie chcesz grac
   OD
   PutE()
   PrintE("Zagraj jeszcze kiedys ponownie.")
   Close(1)      ;zamkniecie K:
RETURN           ;koniec PROCedury glowna
```

Poniższa tabela pokazuje, w jaki sposób program korzysta ze zmiennych. Zawiera ona nazwy zmiennych, ich zasięg i dotępność oraz procedury, które używają tych zmiennych. D oznacza, że zmienna jest dostępna w danej procedurze, a U — że została użyta.

ZMIENNA NAZWA	ZASIEG	PROC rozgrywka	PROC wstep	PROC wynik	FUNC stop	PROC glowna
iloscgier	globalna	D	D	D U	D	D U
iloscprob	globalna	D U	D U	D U	D	D
trafne	globalna	D U	D	D U	D	D
proba	lokalna	D U				
numer	lokalna	D U				
liczab	lokalna	D U				
licznik	lokalna	D U				
znowu	lokalna				D U	
licznik	lokalna		D U			

Widać wyraźnie, że zmienne globalne są dostępne we wszystkich procedurach, podczas gdy zmienne lokalne są dostępne tylko w tych procedurach, w których zostały zadeklarowane. Zwróć uwagę, że są dwie zmienne o nazwie ´licznik´ — jedna w procedurze ´rozgrywka´ i druga w procedurze ´wstęp´. Pomimo takiej samej nazwy nie jest to ta sama zmienna.

Wojciech Zientara



PARAMETRY

Parametry pozwalają na przekazywanie wartości do procedury. Prawdopodobnie zdziwisz się, że jest to konieczne, skoro można użyć zmiennych globalnych zarówno do przekazania wartości do procedury, jak i do wykorzystania w niej. Jest to prawda, lecz istnieją co najmniej dwie przyczyny, dla których stosuje się parametry:

— umożliwia to wykorzystanie jednej procedury do różnych celów;

— pozwala to na manipulowanie wartościami zmiennych wewnątrz procedury bez zmiany wartości zmiennych globalnych.

Omówimy oba te zastosowania oddzielnie, w podanej wyżej kolejności, lecz najpierw konieczne jest podanie formatu listy parametrów.

Parametry w deklaracji PROC lub FUNC

([‹dekl_zmien›] :,‹dekl_zmien›:)

gdzie ‹dekl_zmien› jest normalną deklaracją zmiennej, która jednak nie może zawierać określenia adresu lub wartości początkowej.

Przykłady:

```
PROC Test(BYTE chr,num,i CARD x,y)
INT FUNC DoCommand(INT cmd CARD ptr BYTE
ofset)
CARD FUNC Square(BYTE x)
PROC Jump()
```

Parametry w wywołaniu PROC lub FUNC

([‹wyraż_arytm›] :,‹wyraż_arytm›:)

gdzie ‹wyraż_arytm› jest dowolnym wyrażeniem arytmetycznym.

Przykłady:

```
Test(kot,pies,licznik,2500,$8D00)
sqr=Square(liczba)
Jump()
x=DoCommand(temp,zmiana, A)
```

UWAGA: Każda procedura może mieć maksymalnie osiem parametrów. Użycie większej ich liczby spowoduje błąd podczas kompilacji.

Czas teraz na trochę wyjaśnień. Poniższy przykład pokazuje, jak używać parametrów, i objaśnia pierwszą zaletę stosowania parametrów.

Pokazana poniżej funkcja sprawdza, czy zmienna "znak" typu BYTE jest małą literą. Jeżeli tak, to zwraca odpowiadającą jej dużą literę. W przeciwnym wypadku funkcja zwraca po prostu zmienną "znak". Zauważ, że nigdzie nie deklarujemy "znak". Sposób jej zadeklarowania przedyskutujemy po przykładzie.

```
BYTE FUNC maladuza()

   IF znak>='a AND znak<='z THEN
      RETURN (znak-$20)  ;$20 to roznica
   FI      ;miedzy duzymi i malymi literami
RETURN (znak)     ;w zestawie ATASCII
```

Teraz trzeba zdecydować, gdzie będzie zadeklarowana zmienna "znak". Oczywiście wiesz, że można ją zadeklarować globalnie lub lokalnie. Jeżeli zmienna ta zostanie zadeklarowana lokalnie, to jak nadać jej wartość? Widać więc, że nie można jej zadeklarować lokalnie, gdyż funkcja sama nadawałaby wartość zmiennej, a nie jest to czynność, której od funkcji oczekujemy. Chcielibyśmy, aby wywołanie funkcji "maladuza" było zbliżone do:

znak=maladuza ()

oraz aby funkcja badała "znak" i zwracała dużą literę, gdy to konieczne. Tak więc nie można zadeklarować zmiennej lokalnie. A deklaracja globalna? Teraz funkcja wykona żądaną operację, ponieważ "znak" w wywołaniu funkcji i "znak" w samej funkcji to ta sama zmienna globalna. Jest jednak jedna poważna wada deklarowania "znak" jako zmiennej globalnej. W każdym przypadku, gdy użyjemy funkcji "maladuza", otrzymamy wynik w zmiennej "znak". Jeżeli chcemy wykonać tę funkcję na zmiennej "symbol", musimy postąpić następująco:

```
znak=symbol
znak=maladuza ()
symbol=znak
```

Powoduje to znaczne skomplikowanie programu, gdy chcemy wykonać funkcję na kilkunastu różnych zmiennych. Ponadto, jeśli chcemy wykorzystać procedurę "maladuza" w innym programie, musimy pamiętać o zadeklarowaniu w nim zmiennej globalnej "znak".

A jeśli zadeklarujemy "znak" jako parametr funkcji? Spytacie "Jak...?". Właśnie tak:

```
BYTE FUNC maladuza(BYTE znak)  ;<-
;deklarcja parametru

  IF znak>='a AND znak<='z THEN
    RETURN (znak-$20)
  FI
RETURN (znak)
```

Ale w jaki sposób teraz wywołać funkcję? Bardzo łatwo. Wszystko, co trzeba zrobić, to wprowadzenie zmiennej jako parametru. Na przykład:

```
znak=maladuza (znak)
symbol=maladuza (symbol)
zmienna=maladuza ('a)
```

Zrobienie zmiennej "znak" parametrem funkcji pozwala na użycie tej funkcji do testowania dowolnych zmiennych w dowolnych programach, ponieważ "maladuza" zawsze zastępuje podaną zmienną swoją własną. Nie wykorzystuje ona żadnych zmiennych deklarowanych gdziekolwiek, a jednak pozwala na testowanie zmiennych. W ten sposób ominęliśmy pułapkę deklarowania zmiennej "znak" zarówno globalnie, jak i lokalnie. To właśnie nazwaliśmy zastosowaniem funkcji do różnych celów.

Druga zaleta stosowania parametrów jest znacznie trudniejsza do pokazania, lecz spróbuję to przedstawić tak prosto, jak tylko to możliwe (znów na przykładzie). Poniższa procedura pobiera dwie liczby typu CARD, dzieli pierwszą przez drugą i wyświetla wynik:

```
PROC dzielenie(CARD liczba,dzielnik)

   liczba=/dzielnik
   PrintC(liczba)  ;wyswietlenie wyniku
RETURN
```

A teraz wykorzystanie procedury "dzielenie" w programie:

```
PROC glowna()

   CARD licznik,liczba=[713]

   FOR licznik=1 TO 10
      DO
      PrintC(liczba)
      Print("/")
      PrintC(licznik)
      Print(" = ")
      dzielenie(liczba,licznik)
      PutE()
      OD
RETURN
```

oraz jej rezultat:

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

Zwróć uwagę, że zmienna "numer" pozostaje stała, pomimo iż "liczba" się zmienia. Gdy procedura jest wywoływana, wartość "numer" jest przypisywana "liczbie", lecz "liczba" nie jest z powrotem przypisywana do "numer", gdy procedura jest opuszczana. Gdyby wartość "liczba" była przypisywana z powrotem do "numer", to rezultat byłby następujący:

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

Przekazywanie informacji poprzez parametry jest jednokierunkowe. Informacje mogą być przekazywane do procedury poprzez parametry, lecz nie mogą one być — ogólnie — przekazywane poprzez parametry na zewnątrz procedury. Jeżeli chcesz przekazać pojedyńczą wartość z procedury, to zrealizuj ją w postaci funkcji. Możesz wtedy przekazać wartość z powrotem w instrukcji RETURN. Jeżeli chcesz przekazać z procedury więcej informacji, to musisz użyć zmiennych globalnych lub zastosować wskaźniki jako parametry.

Dopasowanie parametrów

Gdy wywołujesz procedurę, która ma parametry, pierwszy parametr podany w wywołaniu jest przypisywany pierwszej zmiennej z listy parametrów w deklaracji procedury, drugi do drugiej itd. Możesz przekazać mniejszą liczbę parametrów niż wymagana przez procedurę, lecz nigdy większą. Na przykład, jeżeli w deklaracji procedury jest 5 parametrów, to możesz przekazać do tej procedury od zera do pięciu parametrów. Pozwala to na tworzenie procedur, które wymagają różnej liczby parametrów, zależnie od tego, co mają wykonać. W takim przypadku najczęściej wykorzystuje się pierwszy parametr do określenia liczby dalszych parametrów przekazywanych do procedury.

Zgodność parametrów

Jeżeli wartość przekazana jako parametr i wartość oczekiwana przez procedurę są różnych typów, to kompilator zasygnalizuje błąd, gdyż **Action!** wymaga całkowitej zgodności typów parametrów. Na przykład: jeżeli przekażesz do procedury wartość CARD, gdy oczekiwana jest wartość BYTE, to młodszy bajt CARD zostanie przypisany zmiennej BYTE. Dodatkowy bajt (starszy bajt CARD) jest zbędny i kompilator nie wie, co ma z nim zrobić.

Typy parametrów

Dozwolonymi parametrami procedur są zmienne następujących typów:

1) zmienne typów podstawowych;
2) odwołania do tablic, wskaźników i rekordów;
3) nazwy tablic, wskaźników i rekordów.

W trzecim z wymienionych przypadków nazwa jest używana odpowiednio jako wskaźnik pierwszego elementu, wartość lub pierwsze pole w zmiennej, której dotyczy nazwa.

MODULE

Dyrektywa MODULE jest bardzo prostą instrukcją. Jej forma jest:

MODULE

Wskazuje to kompilatorowi, że chcesz zadeklarować jakieś dalsze zmienne globalne. Jest to bardzo użyteczne, gdy piszesz duży program w częściach, z których każda ma swoje zmienne globalne. Jeżeli wpiszesz MODULE na początku każdej części, to kompilator doda wszystkie następujące po nim zmienne do tablicy zmiennych globalnych.

Program nie musi zawierać żadnej dyrektywy MODULE, ponieważ kompilator zakłada, że jedna taka instrukcja znaduje się na początku programu, niezależnie od tego, czy ją tam umieścisz, czy nie.

Deklaracje zmiennych globalnych muszą znajdować się bezpośrednio po dyrektywie MODULE lub na samym początku programu (co oznacza faktycznie umieszczenie ich po MODULE przyjętej przez kompilator).

DYREKTYWY KOMPILATORA

Dyrektywy kompilatora różnią się od zwykłych instrukcji języka tym, że są one wykonywane podczas kompilacji, a nie w czasie wykonywania programu. Instrukcje języka są realizowane po wydaniu w monitorze **Action!** polecenia RUN, a więc wtedy, gdy program przejmuje sterowanie komputerem. Natomiast dyrektywy kompilatora są realizowane, gdy wydasz w monitorze polecenie COMPILE, a więc, gdy sterowanie przejmuje kompilator.

DEFINE

Dyrektywa DEFINE jest bardzo podobna do funkcji zamiany w edytorze (‹SHIFT›‹CONTROL›‹S›), lecz zamiana jest dokonywana podczas kompilacji. Aby to wyjaśnić, trzeba najpierw pokazać format:

DEFINE ‹ident›=‹stała_tekst›[,‹ident›=‹stała_tekst›]

gdzie:

‹ident› jest poprawnym identyfikatorem
‹stała_tekst› jest poprawną stałą tekstową **Action!** (to znaczy ciągiem znaków ograniczonym przez cudzysłowy)

Dyrektywy DEFINE nie są wykorzystywane do tworzenia kodu wynikowego podczas kompilacji, lecz służą do uzyskania czytelniejszej postaci programu źródłowego. Kompilator zastępuje ‹ident›przez ‹stała_tekst›w każdym miejscu programu, w którym zostanie napotkany ‹ident›. Na przykład, gdy kompilujesz program zawierający wiersz

DEFINE rozmiar="256"

kompilator zastępuje w każdym miejscu "rozmiar" przez "256". Pozwala to na zastosowanie wielu ciekawych rozwiązań (i powstanie dodatkowych kłopotów). Ponieważ DEFINE zastępuje każdy ciąg znaków, to możesz również zmienić słowa kluczowe! Jeżeli nie lubisz słowa CARD, możesz zamiast niego stosować słowo RYBA i na początku programu wpisać

DEFINE RYBA="CARD"

Teraz, za każdym razem, gdy podczas kompilacji kompilator napotka słowo "RYBA", pomyśli: „To w rzeczywistości oznacza słowo CARD, więc trzeba wpisać je zamiast RYBA". Dla bliższego poznania tej dyrektywy jeszcze kilka przykładów:

```
DEFINE liston="SET $49A=1"
DEFINE begin="DO", end ="OD"
DEFINE jeden="1"
```

UWAGA: Nie zapomnij, że stała tekstowa musi być obustronnie ograniczona cudzysłowami.

Aby pokazać jeszcze dokładniej, co DEFINE robi, a czego nie robi, posłużymy się tabelką wyjaśniającą efekty działania DEFINE w różnych miejscach programu.

instrukcja	komentarz
DEFINE trzy = "3"	dyrektywa
PrintBE(trzy)	wyświetla '3' i EOL
; trzy stany	zamienia 'trzy' na '3'

```
; trzysta bajtów          nie zmienia 'trzysta'
PrintE("trzy stany")      nie zmienia w cudzy-
                          słowach
```

INCLUDE

Dyrektywa INCLUDE pozwala na włączenie innego programu do programu właśnie kompilowanego. Załóżmy, że masz program o nazwie "IOBLOK.ACT", który realizuje procedury wejścia/wyjścia i chcesz użyć tych procedur w pisanym właśnie programie. Wszystko, co musisz zrobić, to umieszczenie w pisanym programie następującego wiersza:

```
INCLUDE "D1:IOBLOK.ACT"
```

UWAGA: Specyfikacja pliku musi być umieszczona w cudzysłowach.

Powyższa dyrektywa musi być umieszczona w programie przez użyciem jakiejkolwiek procedury, która znajduje się w pliku "IOBLOK.ACT". Zauważ, że przykład zakłada, iż dyskietka z plikiem "IOBLOK.ACT" znajduje się w stacji dysków numer 1. Jeżeli w specyfikacji pliku nie podasz urządzenia, to kompilator przyjmuje, że urządzeniem jest "D1:". Można włączyć plik z dowolnego urządzenia, z którego możliwy jest odczyt (np. "P:" jest nieprawidłowe). Oto kilka dalszych przykładów:

```
INCLUDE "D2:IOLIB.ACT"
INCLUDE "PROG1.DAT"
INCLUDE "C:"
```

UWAGA: Większość systemów operacyjnych wymaga, aby nazwa pliku była podana dużymi literami.

Użyteczną cechą dyrektywy INCLUDE jest możliwość umieszczenia takiej dyrektywy również w programie, który jest włączany przez INCLUDE (czyli możliwość zagnieżdżania). **Action!** pozwala na maksymalne zagnieżdżenie do sześciu poziomów, lecz system operacyjny i urządzenia peryferyjne mają dodatkowe ograniczenia. Gdy zostanie przekroczona liczba zagnieżdżeń dozwolona przez system operacyjny, sygnalizowany jest błąd 161. Magnetofon pozwala na jeden poziom zagnieżdżenia, natomiast stacja dysków na trzy. Jeżeli w edytorze **Action!** nie ma żadnego programu, to liczba dozwolonych zagnieżdżeń jest zmniejszana o jedno.

SET

Dyrektywa SET jest wykorzystywana do modyfikowania zawartości pamięci RAM komputera. W czasie kompilacji SET wpisuje nową wartość do wskazanej komórki pamięci. W większości wypadków dyrektywa ta jest używana do zmiany wariantów pracy edytora i kompilatora przez program użytkownika. Format dyrektywy SET jest następujący:

```
SET ‹adres›=‹wartość›
```

gdzie ‹adres› i ‹wartość› muszą być stałymi kompilatora.

Wynikiem działania dyrektywy SET jest wpisanie ‹wartości› do komórki pamięci ‹adres›. Jeżeli ‹wartość› jest większa niż 255, to jest ona wpisywana do komórek ‹adres› i ‹adres+1›. Jest to spowodowane tym, że 255 ($FF) jest największą liczbą, którą można zmieścić w jednym bajcie, więc każda większa liczba wymaga dwóch bajtów do jej zapisania.

Przykłady:

```
SET $600=64          ;ustawia 64 w komórce $600
SET max=16           ;ustawia max=16
SET 10000=$FFFF      ;ustawia $FFFF w 1000 i 10001
SET $CF00=kot        ;ustawia kot w $CF00 i $CF01
DEFINE adr="$7000"
SET adr=$42
```

Ostatni przykład pokazuje definiowanie przez DEFINE stałej liczbowej użytej w SET. Ponieważ wartość z DEFINE jest w czasie kompilacji stała, to można ją zastosować w dyrektywie SET. Pamiętaj tylko o zdefiniowaniu stałej przed użyciem jej w SET.

UWAGA: Nie należy mylić działania dyrektywy SET podczas kompilacji z podobnym działaniem procedur Poke i PokeC w trakcie pracy programu.

ROZSZERZONE TYPY DANYCH

Rozszerzone typy danych czynią **Action!** językiem bardziej elastycznym niż inne dostępne dla komputerów Atari. Jak instrukcje strukturalne operujące grupami instrukcji prostych zwiększają możliwości języka **Action!**, tak rozszerzone typy danych operują grupami zmiennych typów podstawowych jeszcze bardziej powiększając możliwości języka.

Trzema rozszerzonymi typami danych w **Action!** są:

1) wskaźniki
2) tablice
3) rekordy

Opiszemy je teraz oddzielnie według kolejności na powyższej liście.

WSKAŹNIKI

Słowo "wskaźnik" przypomina rzecz, której używa nauczyciel, aby pokazać coś na mapie. Tak, to jest to. W **Action!** znaczenie wskaźnika jest bardzo podobne. Wskaźnik zawiera adres komórki pamięci, a więc wskazuje komórkę. Możesz zmienić wartość wskaźnika i wskazać nim inną komórkę, tak jak nauczyciel może przesunąć swój wskaźnik na inne miejsce mapy. Najważniejszą różnicą jest to, że wskaźnik nauczyciela pokazuje miasta lub kraje, a wskaźnik w **Action!** wartości BYTE, CARD lub INT.

Deklaracja wskaźnika

Format stosowany do deklarowania wskaźnika jest bardzo podobny do formatu deklaracji zmiennej typu podstawowego. Jedyną różnicą jest zaznaczenie, że chodzi o wskaźnik, a nie o zmienną typu podstawowego.

```
‹typ› POINTER ‹ident›[=‹adres›] :,‹ident›[=‹adres›]:
```

gdzie:

‹typ› jest podstawowym typem danych, na które będzie wskazywał wskaźnik;

POINTER jest słowem kluczowym stosowanym w celu zaznaczenia, że deklarowana zmienna jest wskaźnikiem;

‹ident› jest nazwą zmiennej wskaźnikowej (wskaźnika);

‹adres› określa komórkę pamięci, na którą początkowo wskazuje wskaźnik; musi to być stała kompilatora.

Ponieważ zmienna wskaźnikowa zawiera adres, to musi umożliwiać przechowanie wartości z zakresu od 0 do 65535 (od $0 do $FFFF), gdyż Atari ma tyle oddzielnych komórek pamięci. Aby to zapewnić, wskaźniki są zapisywane jako dwubajtowe liczby bez znaku (w kolejności młodszy, starszy bajt). Oznacza to, że są one zapisywane jako liczby typu CARD, lecz są interpretowane jako adresy.

Ponieważ wykorzystanie wskaźników jest pokazane dalej, to teraz tylko kilka przykładów deklaracji wskaźników:

```
BYTE POINTER wsk      ;deklaruje wsk jako
                      ;wskaźnik wartości
                      ;typu BYTE
CARD POINTER adr      ;deklaruje adr jako
                      ;wskaźnik wartości
                      ;typu CARD
INT POINTER ip=$8000  ;deklaruje ip jako wska-
                      ;źnik wartości
                      ;typu INT i ustawia go
                      ;na wskazywanie
                      ;adresu $8000
```

Operowanie wskaźnikami

Wskaźniki w **Action!** mogą być wykorzystane do operowania różnymi obiektami, gdyż można je łatwo ustawić na wskazywanie różnych komórek pamięci. Bardzo ułatwia to katalogowanie i tablicowanie informacji.

Poniższy program jest prostym przykładem, który pokazuje ideę działania wskaźników. Wprowadzony jest w nim operator adresowy "^", który zostanie dokładniej opisany po przykładzie.

Przykład:

```
PROC uzyciewskaznika()

  BYTE numer=$E0,    ;deklaruje dwie
       znak=$E1      ;zmienne typu BYTE
  BYTE POINTER bwsk ;deklaruje wskaznik
                    ;zmiennej typu BYTE

  bwsk=@numer       ;bwsk wskazuje numer
  Print("bwsk wskazuje teraz adres ")
  PrintF("%H",bwsk) ;druk adresu numer
  PutE()
  bwsk^=255         ;wpisanie 255 do
                    ;wskazywanej komorki
  Print("numer jest teraz rowny ")
  PrintBE(numer)    ;pokazuje numer=255
  bwsk^=0           ;wpisanie 0 do numer
  Print("numer jest teraz rowny ")
  PrintBE(numer)    ;teraz numer=0
  bwsk=@znak        ;bwsk wskazuje znak
  Print("bwsk wskazuje teraz adres ")
  PrintF("%H",bwsk) ;druk adresu znak
  PutE()
  bwsk^='q          ;zmienia znak na 'q
  Print("znak jest teraz rowny ")
  Put(znak)         ;pokazuje znak='q
  PutE()
  bwsk^='z          ;zmienia znak na 'z
  Print("znak jest teraz rowny ")
  Put(znak)         ;pokazuje znak='z
  PutE()
RETURN
```

Wynik:

```
bwsk wskazuje teraz adres $E0
numer jest teraz rowny 255
numer jest teraz rowny 0
bwsk wskazuje teraz adres $E1
znak jest teraz rowny q
znak jest teraz rowny z
```

Zwróć uwagę, że używamy operatora "^", gdy chcemy umieścić wartość w miejscu wskazywanym przez wskaźnik. Tak więc wiersz "bwsk^=0" w powyższym przykładzie jest równoznaczny z "numer=0", ponieważ "bwsk" wskazuje w tym czasie na "numer". Pomimo iż nie użyliśmy tego w przykładzie, odwołania poprzez wskaźniki mogą być także stosowane w wyrażeniach arytmetycznych:

```
x=wsk^
```

Zauważ również, że poprawne jest "PrintF("%H,bwsk)". Oznacza to więc, że "bwsk" może być dostępny zarówno jako adres, jak i wartość typu CARD. Jest to bardzo pomocne przy usuwaniu błędów w programie, gdyż łatwo można sprawdzić, na co aktualnie wskazuje wskaźnik.

TABLICE

Tablice pozwalają na operowanie listami zmiennych przez umożliwienie dostępu do każdej zmiennej z listy przy użyciu tylko nazwy tablicy i indeksu. Zmienne w liście muszą być tego samego typu, a dozwolone są tylko typu podstawowe. Nazwa tablicy wskazuje listę zmiennych, indeks zaś określa konkretny element tej listy. Indeks jest po prostu liczbą, więc odwołanie brzmi: "Potrzebny jest n-ty element tablicy x", gdzie n jest indeksem, a x nazwą tablicy.

Deklaracja tablicy

Deklarowanie tablicy w **Action!** jest proste. Są jednak różne warianty, które pozwalają na zadeklarowanie różnych parametrów tablicy, w tym adresu, rozmiaru i początkowej zawartości. Ze względu na te warianty najpierw zobaczmy format w postaci skondensowanej.

```
‹typ› ARRAY ‹wyrt_pocz› :,‹wart_pocz›:
```

gdzie:

‹typ› jest podstawowym typem danych będących elementami tablicy;

ARRAY jest słowem kluczowym określającym tablicę;

‹wart_pocz› jest informacją wymaganą do zadeklarowania zmiennej jako tablicy elementów typu ‹typ›.

‹wart_pocz› ma format:

‹ident›[(rozmiar)][=‹adres› [‹wartości›] ‹st_tekst›]
gdzie:
‹ident› jest nazwą zmiennej (tablicy);
‹rozmiar› jest rozmiarem tablicy i musi być stałą liczbową;
‹adres› określa adres pierwszego elementu tablicy i musi być stałą kompilatora;
[‹wartości›] określa początkowe wartości elementów tablicy, każda wartość musi być stałą liczbową;
‹st_tekst› określa początkowe wartości elementów tablicy według ciągu znaków, przy czym pierwszy element będzie długością tego ciągu.

Teraz czas na kilka przykładów, które pomogą zrozumieć sposób deklarowania tablic.

BYTE ARRAY a,b ;deklaruje dwie tablice z elementami typu ;BYTE, bez określenia ich rozmiaru
INT ARRAY x(10) ; deklaruje tablicę INT zawierającą 10 ;elementów
BYTE ARRAY txt="To jest stała tekstowa" ;deklaruje tablicę ;BYTE i wypełnia ją stałą tekstową
CARD ARRAY tab=$8000 ;deklaruje tablicę CARD, która ;rozpoczyna się od adresu $8000 i nie ma ;określonego rozmiaru
BYTE ARRAY test=[4 7 18] ;deklaruje tablicę BYTE i ;wypełnia ją wartościami

UWAGA: Powinieneś określać wymiar deklarowanej tablicy zawsze, gdy jest to możliwe, lecz są przypadki, gdy nie można lub nie trzeba tego robić:
— gdy nie wiesz, jak duża będzie tablica (np. przy definiowaniu parametrów procedury, gdy nie jest znany rozmiar tablicy przekazywanej jako parametr),
— gdy wypełniasz tablicę wartościami początkowymi (zarówno przy pomocy [‹wartości›], jak i ‹st_tekst› i nie zamierzasz nic do niej dodawać.

Pamiętaj ponadto, że pierwszy bajt stałej tekstowej zawiera wartość określającą długość ciągu. W celu zwiększenia długości ciągu trzeba więc najpierw zmienić bajt długości, który jest zerowym elementem tablicy.

Reprezentacja wewnętrzna

Wewnętrzna reprezentacja tablic jest bardzo podobna do wskaźników. Jest tak, ponieważ nazwa tablicy jest rzeczywiście wskaźnikiem pierwszego elementu tablicy. Tablica jest po prostu ciągłą grupą pól, z których każdy zawiera jeden element tablicy. Rozmiar pola jest określony przez typ danych w tablicy i wynosi 1 bajt dla tablic typu BYTE oraz dwa bajty dla tablic CARD i INT.

Operowanie tablicami

Użycie tablic i operowanie nimi nie jest trudne, gdy tylko wiesz, jak deklarować tablice i jak się odwoływać do ich elementów. Deklarowanie tablic już zostało opisane, więc teraz zajmiemy się odwołaniami do elementów.
Przykład 1:

```
PROC testodwolan()
  BYTE x
  BYTE ARRAY cyfry(10)

  FOR x=0 TO 9 ;tablica cyfry ma 10
               ;elementow, wi c indeksy
               ;sa liczone od 0 do 9
  DO
    cyfry(x)=x+'A ;x-ty element tablicy
                  ;uzyskuje wartosc x+'A
    Put(cyfry(x)) ;drukuje x-ty element
                  ;jako znak
    Print(" ")  ;odstep miedzy znakami
  OD
  PutE()
RETURN
```

Wynik 1:

```
A B C D E F G H I J
```

W powyższym programie znajdują się dwa odwołania do tablicy — "cyfry(x)" w instrukcji przypisania i "cyfry(x)" jako parametr procedury bibliotecznej Put. Te i wszystkie inne odwołania mają format:
‹ident›(‹indeks›)
gdzie:
‹ident› jest nazwą tablicy, do której się odwołujesz;
‹indeks› jest numerem elementu w tej tablicy i jest wyrażeniem arytmetycznym.

Jak wyjaśniono w komentarzu przy instrukcji FOR, indeksy tablic nie rozpoczynają się od 1. Pierwszym elementem tablicy "koty" jest "koty(0)", a nie "koty(1)". Może się to wydawać dziwne, lecz szybko to zapamiętasz.
Przykład 2:

```
PROC zmianatablicy()

  BYTE ARRAY btab

  btab="To jest lancuch 1."
  PrintC(btab)    ;btab jako liczba CARD
  Print("  ")
  PrintE(btab)    ;btab jako tablica
  btab="To jest lancuch 2."
  PrintC(btab)
  Print("  ")
  PrintE(btab)
RETURN
```

Wynik 2:

```
10352  To jest lancuch 1.
10414  To jest lancuch 2.
```

KOMENTARZ: Wyraźnie widać, że zmiana uległ adres, na który wskazuje nazwa "btab". Ponowne przypisanie całej tablicy (gdy jest wykonywane przy użyciu stałych tekstowych) nie powoduje przepisania nowego ciągu znaków do obszaru pamięci zajętego przez stary, lecz tylko przypisuje tablicy nowe miejsce w pamięci. Powoduje to zmianę wartości "btab" tak, aby wskazywała początkowy adres nowego ciągu. Stary ciąg nadal pozostaje w pamięci, lecz nie jest niczym wskazywany, więc jest niedostępny.

Zwróć uwagę, że "PrintE(btab)" jest poprawne, ponieważ "btab" wskazuje na poprawną stałą tekstową, której typ jest zgodny z typem parametru wymaganego przez biblioteczną procedurę PrintE.
Przykład 3:

```
PROC rownetablice()

  BYTE ARRAY a="Oto stala tekstowa",
              btab

  btab=a
  PrintE(a)
  PrintE(btab)
RETURN
```

Wynik 3:

```
Oto stala tekstowa
Oto stala tekstowa
```

KOMENTARZ: Wszystko, co robi ten program, to pokazanie, że dwóm tablicom można przypisać tę samą wartość przez proste ustawienie wskaźnika tablicy na ten sam adres w pamięci. W powyższym przypadku jest to stała tekstowa, na którą wskazują obie nazwy tablic.

Mogłeś zauważyć, że nie zastosowaliśmy konstrukcji podobnej do:

```
BYTE ARRAY a=['T 'o '  's 't 'a 'l 'a]

PrintE(a)
```

Taka konstrukcja nie działa prawidłowo. Pamiętaj, że stała tekstowa różni się od zwykłego ciągu, gdyż jej pierwszy bajt określa jej długość. Procedury, które oczekują stałej tekstowej, nie działają więc, gdy podasz im coś innego.

Wojciech Zientara

Na zakończenie prezentujemy przykładowy program, który wykorzystuje wszystkie, uprzednio opisane zastosowania tablic.

Przykład 4:
SCENARIUSZ: Masz program, który podaje tylko numer błędu, gdy użytkownik popełni omyłkę i chcesz, aby także wyświetlał on komunikat błędu. Można to zrobić przy pomocy tablic, jak w poniższym programie. Opis jego działania jest podany później.

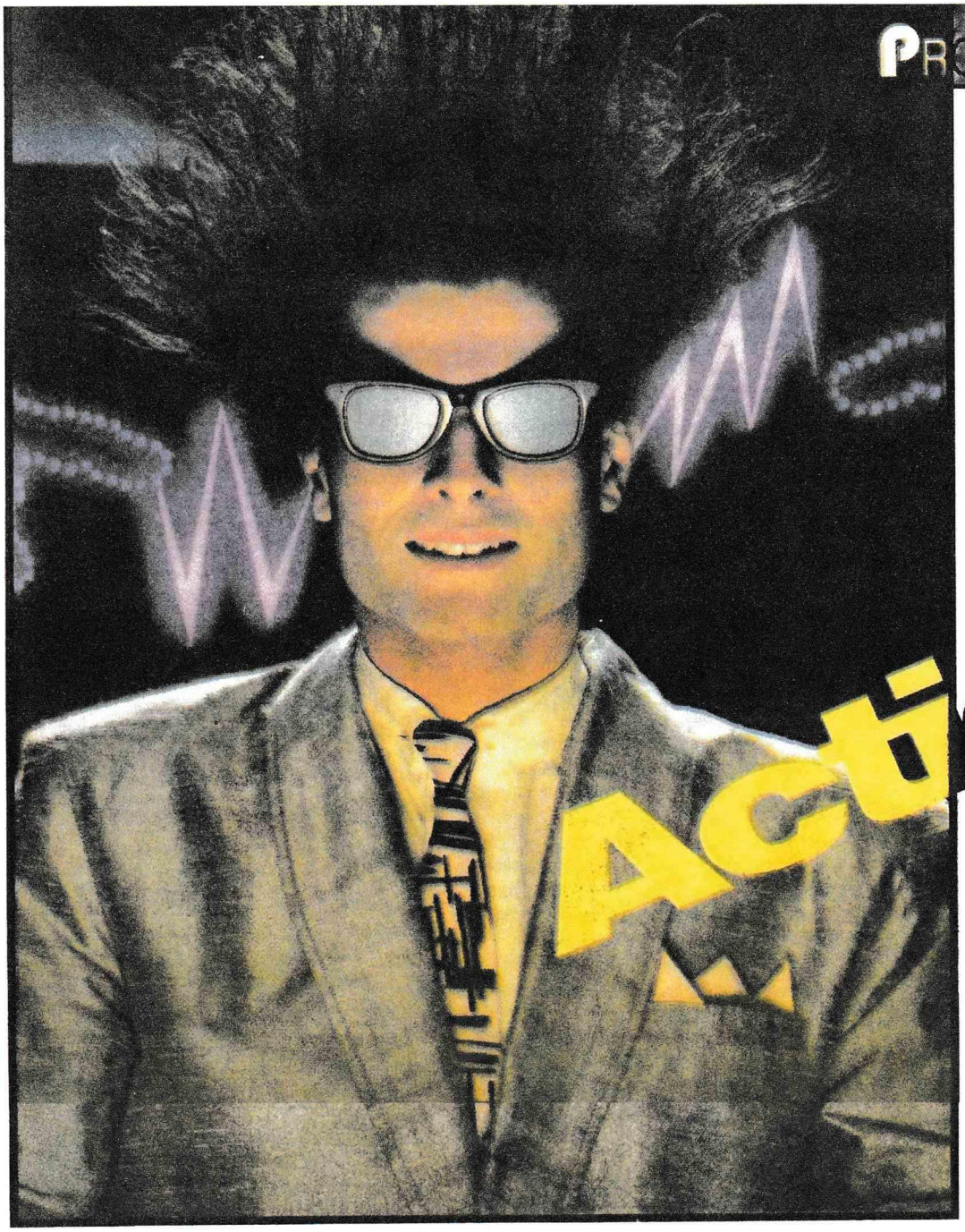
```
PROC bledy(BYTE numbl)
;**** Ta procedura odczytuje numer
;bledu i wyswietla odpowiedni
;komunikat. Opis jej dzialania
;jest umieszczony ponizej.

  BYTE ARRAY kombl ;komunikaty

  CARD ARRAY adr(6) ;zawiera adresy
                    ;komunikatow

  adr(0)="Niedozwolony rozkaz"
  adr(1)="Niedozwolony znak"
  adr(2)="Zla nazwa pliku"
  adr(3)="Liczba zbyt duza"
  adr(4)="Zly typ liczby"
  adr(5)="Nieznany blad"
  kombl=adr(numbl)
  Print("BLAD #")
  PrintB(numbl)
  Print(":  ")
  PrintE(kombl)
  PutE()
RETURN ;koniec procedury 'bledy'

PROC glowna()
;**** Ta procedura sluzy tylko do
;wywolania procedury 'bledy' przy
;uzyciu wszystkich poprawnych
;numerow bledow, aby pokazac
;dzialanie tablicy.

  BYTE blad

  FOR blad=0 TO 5
    DO
      bledy(blad)
    OD
RETURN ;koniec procedury 'glowna'
```

Wynik:

```
BLAD #0:  Niedozwolony rozkaz
BLAD #1:  Niedozwolony znak
BLAD #2:  Zla nazwa pliku
BLAD #3:  Liczba zbyt duza
BLAD #4:  Zly typ liczby
BLAD #5:  Nieznany blad
```

KOMENTARZ: Sposób, w jaki wypełniona została tablica CARD w powyższym przykładzie, jest dziwny (jak można wypełnić tablicę CARD elementami ciągu?), lecz jest on poprawny, gdyż do elementu tablicy nie jest przypisywana sama stała tekstowa, ale jej adres. Czyni to każdy element tablicy ukrytym wskaźnikiem ciągu. Wszystkim, co trzeba uczynić, jest przypisanie wartości odpowiedniego elementu tablicy (czyli wskazującego właściwy komunikat błędu) do tablicy 'kombl', a więc uczynienie 'kombl' wskaźnikiem właściwego komunikatu. Następnie trzeba tylko wyświetlić komunikat.

Pokazany wyżej przykład jest bardzo trudny. Jego zrozumienie wymaga dokładnego zrozumienia istoty tablic i ich wewnętrznej reprezentacji. Został on jednakże umieszczony tu w celu pokazania możliwości, jakie udostępnia wykorzystanie tablic.



Action! (5)

REKORDY

Rekordy są konstrukcjami, które pozwalają na grupowanie pewnych informacji, które są w pewien sposób zależne, choć nie są tego samego typu. Przykładem rekordu może być prawo jazdy. Zawiera ono imię, nazwisko, adres, fotografię i numer prawa jazdy. Informacje te w pewien sposób opisują właściciela, lecz są one różnych typów. Imię i nazwisko są ciągami znaków, fotografia jest obrazem, zaś adres i numer są złożone zarówno z liter, jak i cyfr. Oczywiście, język **Action!** nie może wykorzystywać wszystkich tych typów. Zamiast nich rekord może grupować informacje typów zrozumiałych dla kompilatora, czyli podstawowych typów danych.

Deklaracja rekordu

Rekordy w **Action!** operują podstawowymi typami danych przez tworzenie nowych typów złożonych z jednego lub kilku typów podstawowych. Zmienne tak utworzonych typów deklaruje się następnie tak, jak zmienne typów BYTE, INT lub CARD. Pozwala to na deklarowanie dowolnej liczby zmiennych jednego typu określonego przez rekord, lecz bez możliwości zmiany formatu rekordu.

Następny ustęp pokazuje tworzenie nowych typów danych, a kolejny demonstruje deklarowanie zmiennych o uprzednio określonym typie rekordu.

Deklaracja TYPE

Bez dalszych ceregieli należy teraz zaprezentować formę stosowaną do deklarowania rekordu:

TYPE ‹ident›=[‹dekl_zmienn›]

gdzie **TYPE** jest słowem kluczowym oznaczającym deklarację rekordu;

‹ident› jest nazwą deklarowanego typu (rekordu);

‹dekl_zmienn› są poprawnymi deklaracjami zmiennych, lecz bez określenia ich początkowych wartości.

W zrozumieniu tego formatu pomoże poniższy przykład:

```
TYPE rekord=[BYTE b1,b2      ; dwa pola typu BYTE
             INT i1          ; jedno pole INT
             CARD c1,c2,c3   ; trzy pola CARD
             BYTE b3]        ; ostatnie pole także
                             ; BYTE
```

Wymaga to jeszcze pewnych wyjaśnień, które podam punkt po punkcie:

TYPE rekord

Definiujemy nowy typ danych o nazwie „rekord".

BYTE b1,b2

Dwa pierwsze pola tego typu są polami typu BYTE i mają nazwy „b1" i „b2".

INT i1

Trzecie pole jest typu INT i nosi nazwę „i1".

CARD c1,c2,c3

Pola od czwartego do szóstego są typu CARD i mają odpowiednio nazwy „c1", „c2" i „c3".

BYTE b3

Siódme i ostatnie pole rekordu „rekord" jest typu BYTE i nazywa się „b3".

Zwróć uwagę, że nie ma przecinków między deklaracjami różnych zmiennych (np. między deklaracjami BYTE i CARD). Jeżeli umieścisz tam przecinek, kompilator odczyta słowo określające typ podstawowy (CARD, BYTE, INT) jako nazwę zmiennej, co spowoduje błąd.

Deklarowanie zmiennych

Zobaczyłeś już, jak zadeklarować typ rekordowy, czas więc teraz na pokazanie, jak zadeklarować zmienną danego typu. Format jest bardzo podobny do stosowanego przy deklarowaniu zmiennych typów podstawowych:

‹ident› ‹zmienna› [=‹adr›][:, ‹zmienna›[=‹adr›]:

gdzie ‹ident› jest nazwą typu rekordowego;
‹zmienna› jest zmienną, której typ jest deklarowany jako rekord;
‹adr› jest adresem w pamięci, pod którym chcesz umieścić zmienną — musi to być stała liczbowa.

Poniżej znajduje się przykład wykorzystujący pokazaną poprzednio deklarację rekordu. Po przykładzie następuje jego wyjaśnienie.

```
TYPE rekord=[BYTE b1,b2          ; ta sama deklaracja,
            INT i1               ; TYPE, która była
            CARD c1,c2,c3        ; użyta poprzednio
            BYTE b3]
rekord rek1,                     ; deklaruje rek1
                                 ; jako typ „rekord"
rek2=$8000                       ; deklaruje rek2
                                 ; jako typ „rekord"
                                 ; i umieszcza ją
                                 ; pod adresem
                                 ; $8000
```

rekord
Wskazuje, że następujące później zmienne są danymi typu „rekord", jak BYTE, INT i CARD (gdy są użyte w deklaracji) wskazują dane tych typów.

rek1
Deklaruje „rek1" jako zmienną typu „rekord".

rek2=$8000
Deklaruje „rek2" jako zmienną typu „rekord" i umieszcza ją w pamięci od adresu $8000.

Operowanie rekordami

Aby nauczyć się operowania rekordami, najpierw musisz nauczyć się odwoływania do pól wewnątrz rekordu. Poniższy program wykonuje to przy użyciu operatora „." (kropka). Jego użycie jest opisane po programie.

```
PROC odwolanie()
;*** Procedura odczytuje informacje
;o pracowniku i wyswietla je dla
;sprawdzenia poprawnosci.

  TYPE idinfo=[BYTE poziom
               CARD idnum,rok]
  idinfo rec ;zmienna 'rec' jest
             ;typu 'idinfo'

  Print("Twoj numer identyfikacyjny? ")
  rec.idnum=InputC()
  Print("Poziom pracy (A-Z)? ")
  rec.poziom=GetD(7)
  PutE()
  Print("Rok rozpoczecia pracy? ")
  rec.rok=InputC()
  PutE()
  PrintE("O.K. Wpisano:")
  PutE()
  Print("I.D. #  ")
  PrintCE(rec.idnum)
  Print("Poziom:  ")
  Put(rec.poziom)
  PutE()
  Print("Rok:  ")
  PrintCE(rec.rok)
RETURN  ;koniec procedury 'odwolanie'
```

Wynik:

```
Twoj numer identyfikacyjny? 4365
Poziom pracy (A-Z)? L
Rok rozpoczecia pracy? 1988

O.K. Wpisano:

I.D. #  4365
Poziom:  L
Rok:  1988
```

Kropka (.) jest używana do wskazania kompilatorowi, że wykonujesz odwołanie do rekordu (i jest to jedyne poprawne odwołanie do rekordu). W powyższym przykładzie programie mogłeś zauważyć, że format odwołania do rekordu jest następujący:

‹nazwa_rekordu›.‹nazwa_pola›

Zwróć uwagę, że ‹nazwa_pola› i ‹nazwa_rekordu› są określone w różnych deklaracjach, jak uprzednio pokazano. ‹nazwa_pola› jest definiowana w deklaracji TYPE, gdy określasz pola rekordu. Natomiast ‹nazwa_rekordu› jest definiowana w deklaracji zmiennej, gdy określasz zmienne typu rekordowego.

ZASTOSOWANIE ROZSZERZONYCH TYPÓW DANYCH

Rozszerzone typy danych wydają się być ograniczone przez fakt, że mogą one operować tylko typami podstawowymi. Oznacza to, że nie można uzyskać tablicy rekordów, pola tablicowego w rekordzie itd. Jednakże, istnieją sposoby ominięcia tych ograniczeń, jak to pokazuje przykład 4 w opisie tablic („Moje Atari" 2/90). W przykładzie tym tworzymy tablicę wskaźników przy użyciu elementów tablicy CARD, jako wskaźników a nie liczb. W tym ustępie zademonstruję inne sposoby wykorzystania rozszerzonych typów danych, włącznie z programem wykorzystującym rekordy z polami tablicowymi i program, który używa tablicy rekordów.

„Przecież mówiliśmy, że jest to niedozwolone." Jest to niedozwolone, gdy zostanie wykonane bezpośrednio, lecz — jak już wspomniałem — istnieją sposoby ominięcia dosłownej definicji typu rozszerzonego.

W kolejnym przykładzie niezwymiarowana tablica jest wypełniana listą rekordów. Gdy zdefiniujemy rekord „wirtualny" (wyobrażony), to sposób realizacji jest bardzo prosty, ponieważ tablica jest teraz tablicą BYTE zawierającą bloki bajtów pogrupowane w rekordy wirtualne.

Rekord wirtualny nie jest rekordem w takim sensie, jak deklarowany typ rekordowy. Jest on rekordem tylko dlatego, że umożliwia dostęp do pamięci tak, jakby był to rekord, choć w rzeczywistości jest to tylko ciąg bajtów. Wypełniamy więc tylko tablicę BYTE w taki sposób, że wygląda ona jak ciąg rekordów, a nie bajtów. Jest to zrealizowane przez zadeklarowanie typu rekordowego, a następnie zadeklarowanie wskaźników tego typu. Później operujemy tablicą w blokach o rozmiarze jednego rekordu przez wykorzystanie wskaźnika o wartości zmienianej o długości jednego rekordu.

Przykład 1:

```
MODULE

  TYPE idinfo=[CARD idnum,kod
               BYTE poziom]

  BYTE ARRAY idtab(1000)

  DEFINE rozmiar="5"

  CARD licznik=[0]

PROC wypelnienie()
;**** Procedura pobiera informacje
;umieszczajac je w tablicy rekordow
;przy uzyciu wskaznika rekordu oraz
;indeksowania wskaznikow w tablicy.

  idinfo POINTER nowy

  BYTE dalej

  DO
    nowy=idtab+(licznik*rozmiar)
    Print("Numer I.D.?  ")
    nowy.idnum=InputC()
    Print("Poziom pracy (A-Z)?  ")
    nowy.poziom=GetD(7)
    PutE()
    Print("Kod dostepu?  ")
    nowy.kod=InputC()
    licznik==+1
    PutE()
    Print("Kolejny rekord (T lub N)?")
    dalej=GetD(7)
    PutE() PutE()
    UNTIL dalej='N OR dalej='n
  OD
RETURN
```

UWAGA: Procedura ta nie sprawdza, czy znajdujesz się jeszcze w obrębie tablicy, również sam **Action!** tego nie kontroluje. Procedurę sprawdzającą aktualny rozmiar zajętej części tablicy możesz dopisać samodzielnie.

Szczegółowego wyjaśnienia wymagają pewne fragmenty tej procedury, włącznie z następującymi wierszami:

```
DEFINE rozmiar="5"
idinfo POINTER nowy
nowy=idtab+(licznik*rozmiar)
nowy.XXX=xxx
licznik==+1
```

Chodzi tu nie tylko o objaśnienie samych instrukcji, lecz także o wytłumaczenie sposobu użytego do uzyskania tablicy rekordów.

DEFINE rozmiar="5"
Dyrektywa DEFINE jest użyta dla określenia rozmiaru skoku niezbędnego do przejścia tablicy. Rekord typu 'idinfo' ma długość 5 bajtów (dwie liczby CARD i jedną BYTE), więc umożliwia to przechodzenie tablicy pięciobajtowymi krokami. Każdy taki krok powoduje przeskoczenie jednego rekordu, a więc eliminuje możliwość zapisania nowego rekordu na części poprzedniego.

idinfo POINTER nowy
Tu definiujemy wskaźnik typu 'idinfo'. Można wypełnić pola wirtualnego rekordu w tablicy po prostu przez wskazanie wskaźnikiem pierwszego pola w jednym z wirtualnych rekordów, a następnie użycie wskaźnika w odwołaniu do rekordu dla uzyskania dostępu do pojedynczego pola.

nowy=idtab+(licznik*rozmiar)
Przypisanie to powoduje wskazanie przez wskaźnik końca tablicy. Czyni ono to przez dodanie rozmiaru tablicy zajętego przez dotychczasowe rekordy do początkowego adresu tablicy. Zajęta część tablicy jest zwykłym iloczynem liczby rekordów ('licznik') i rozmiaru jednego rekordu ('rozmiar').

nowy.XXX=xxx
'XXX' jest jedną z nazw pól w rekordzie, zaś 'xxx' jest odpowiednią funkcją wejścia, która jest wykorzystywana do wypełnienia tablicy. Ponieważ 'nowy' wskazuje na koniec tablicy, to wypełniany jest nowy rekord. Użycie wskaźnika w odwołaniu do rekordu jest możliwe, gdyż został on zadeklarowany jako wskaźnik tego typu rekordowego.

licznik==+1
Tu po prostu zwiększamy wartość zmiennej, która przechowuje liczbę rekordów znajdujących się aktualnie w tablicy.

W przykładzie 4 użyjemy tej wypełnionej tablicy do weryfikowania informacji wpisanych przez kogoś chcącego uzyskać dostęp do chronionego obszaru (przez sprawdzenie indywidualnego kodu). Trzeba przy tym pamiętać, aby tablicę traktować jako tablicę rekordów i używać tego samego formatu, który był zastosowany przy jej wypełnianiu, gdyż inaczej mogą wystąpić poważne problemy.

Zanim przejdziemy do programu sprawdzającego wypełnioną tablicę, najpierw nieco zmodyfikujemy rekord. Dodamy do niego jeszcze pole, które zawiera personalia pracownika w formie:

nazwisko, imię

Aby to zrealizować trzeba dodać pole tablicowe. Ale, czy na pewno? Zamiast pola tablicowego dodamy na końcu rekordu pole BYTE i zmienimy rozmiar rekordu w dyrektywie DEFINE. Jeżeli zwiększymy go o 20, to uzyskamy 25 bajtów zarezerwowanych dla sześciu bajtów (2 CARD i 2 BYTE). W wolnym miejscu można zapisać ciąg, przez dostęp do ostatniego pola (nowe pole BYTE) i umieszczenie w nim ciągu zamiast bajtu. Ciąg ten nie może być dłuższy niż 19 znaków, ponieważ pierwszy bajt ciągu określa jego

długość. Zobaczmy teraz, jak wygląda rozszerzona wersja naszej procedury.
Przykład 2:

```
MODULE

  TYPE idinfo=[CARD idnum,kod
              BYTE poziom,
                   nazwisko]

  BYTE ARRAY idtab(1000)

  DEFINE rozmiar="25",
         ofset="5"
  CARD licznik=[0]
PROC wypelnienie()
;**** Jest to nieco zmodyfikowana
;wersja poprzedniego przykladu.

  idinfo POINTER nowy

  BYTE POINTER nwsk

  BYTE dalej
  DO
    nowy=idtab+(licznik*rozmiar)
    Print("Numer I.D.?  ")
    nowy.idnum=InputC()
    Print("Poziom pracy (A-Z)?  ")
    nowy.poziom=GetD(7)
    PutE()
    Print("Kod dostepu?  ")
    nowy.kod=InputC()
    nwsk=nowy+ofset
    PrintE("Nazwisko i imie?")
    InputS(nwsk)
    licznik==+1
    PutE()
    Print("Kolejny rekord (T lub N)?")
    dalej=GetD(7)
    PutE() PutE()
    UNTIL dalej='N OR dalej='n
  OD
RETURN
```

Tak jak w poprzednim programie, są tu wiersze wymagające dokładniejszego wyjaśnienia, między innymi:

ofset=„5"
BYTE POINTER nwsk
nwsk=nowy+ofset
InputS(nwsk)

Przed opisem poszczególnych wierszy, jeszcze kilka słów o sposobie umieszczenia nazwiska w tablicy rekordów. Najpierw trzeba znaleźć miejsce zapisania nazwiska, a następnie konieczne jest opracowanie sposobu jego odczytania. Wyjaśnienie wymienionych wyżej instrukcji pokaże, jak zostało to zrealizowane.

ofset=„5"

Definiuje to odstęp w pojedynczym rekordzie od jego początku do pierwszego bajtu ciągu i jest używane do ustawienia wskaźnika ciągu we właściwym miejscu.

BYTE POINTER nwsk

Wskaźnik ten jest używany do wskazania pierwszego bajtu pola 'nazwisko' w rekordzie.

nwsk=nowy+ofset

Jest to ustalenie wartości (czyli miejsca wskazywanego przez wskaźnik) wskaźnika 'nwsk'. Jest ona ustalana przez dodanie do adresu początku rekordu ('nowy') odstępu do pierwszego bajtu ciągu.

InputS(nwsk)

Odczytuje nazwisko i wykorzystuje 'nwsk' jako wskaźnik miejsca jego zapisania, jak to było pokazane wcześniej, lecz zamiast nazwy tablicy użyty jest tu jej wskaźnik (który wskazuje jej pierwszy element).

Gdy mamy już sposób zapisania rekordu w tablicy, to potrzebny jest jeszcze sposób jego odszukania według podanego wzoru. Poniżej pokazana jest funkcja zaprojektowana w tym celu. Uzyskuje ona dostęp do tablicy rekordów z przykładu 2 i zwraca adres pierwszego rekordu, który ma 'idnum' zgodny z podanym parametrem. Jeżeli nie ma takiego rekordu, to zwracane jest zero. Zauważ, że funkcja ta wykorzystuje zmienne zadeklarowane globalnie (czyli po MODULE) w poprzednim przykładzie.
Przykład 3:

```
CARD FUNC szukaj(CARD testnum)

  idinfo POINTER wsk

  BYTE licz

  FOR licz=0 TO (licznik-1)
    DO
      wsk=idtab+(licz*rozmiar)
      IF wsk.idnum=testnum THEN
        RETURN(wsk)
      FI
    OD
RETURN(0)
```

Wszystko, co wykonuje ta funkcja, to tylko odczyt każdego rekordu i porównanie jego pola 'idnum' z podaną wartością 'testnum'. Teraz przy pomocy dwóch ostatnich przykładów zbudujmy cały program.
Przykład 4:

```
MODULE

  TYPE idinfo=[CARD idnum,kod
              BYTE poziom,
                   nazwisko]

  BYTE ARRAY idtab(1000)

  DEFINE rozmiar="25",
         ofset="5"

  CARD licznik=[0]

PROC wypelnienie()
;**** Znowu procedura wypelniania.

  idinfo POINTER nowy

  BYTE POINTER nwsk

  BYTE dalej

  DO
    nowy=idtab+(licznik*rozmiar)
    Print("Numer I.D.?  ")
    nowy.idnum=InputC()
    Print("Poziom pracy (A-Z)?  ")
    nowy.poziom=GetD(7)
    PutE()
    Print("Kod dostepu?  ")
    nowy.kod=InputC()
    nwsk=nowy+ofset
    PrintE("Nazwisko i imie?")
    InputS(nwsk)
    licznik==+1
    PutE()
    Print("Kolejny rekord (T lub N)?")
    dalej=GetD(7)
    PutE() PutE()
    UNTIL dalej='N OR dalej='n
  OD
RETURN

CARD FUNC szukaj(CARD testnum)

  idinfo POINTER wsk

  BYTE licz

  FOR licz=0 TO (licznik-1)
    DO
      wsk=idtab+(licz*rozmiar)
      IF wsk.idnum=testnum THEN
        RETURN(wsk)
      FI
    OD
RETURN(0)
PROC baza()

  idinfo POINTER rwsk

  BYTE POINTER nwsk

  CARD id_num,kod_num,
       klucz=[65535]

  BYTE tryb

  PrintE("Start...")
  PrintE("Wybierz tryb pracy:")
  PrintE("R = rozszerzenie listy")
  PrintE("T = testowanie")
  Print(">> ")
  tryb=GetD(7)
  Put(tryb) PutE()
  IF tryb='T OR tryb='t THEN
    wypelnienie()
  ELSE
    DO
      Print("Numer I.D. >> ")
      id_num=InputC()
      IF id_num=klucz THEN
        EXIT
      ELSE
        rwsk=szukaj(id_num)
        IF rwsk=0 THEN
          PrintE("BRAK DOSTEPU.")
        ELSE
          Print("Kod >>  ")
          kod_num=InputC()
          IF rwsk.kod=kod_num THEN
            nwsk=rwsk+ofset
            Print("I.D. #  ")
            PrintCE(rwsk.idnum)
            Print("Poziom:  ")
            Put(rwsk.poziom)
            PutE()
            Print("Nazwisko:  ")
            PrintE(nwsk)
            PutE()
            PrintE("O.K.")
          ELSE
            PrintE("BRAK DOSTEPU.")
          FI
        FI
      FI
    OD
  FI
  PrintE("System zamkniety...")
RETURN
```

ZAAWANSOWANE TECHNIKI PROGRAMOWANIA

Ta część prezentuje pewne techniki programowania, które mogą być bardzo użyteczne dla zaawansowanych programistów. Dotychczas ograniczaliśmy nasz opis **Action!** do samego języka, bez odniesień do komputera. Większa część tego rozdziału jest poświęcona współpracy **Action!** z informacjami znajdującymi się poza nim, w tym również z procedurami systemu operacyjnego i zmiennymi systemowymi.

Bloki kodu

Bloki kodu pozwalają na dołączenie języka maszynowego do programu. Gdy kompilator napotka blok kodu, to umieszcza zawarte w nim wartości w generowanym programie wynikowym tak, jakby były to normalne wartości tworzone przez kompilator. Nie jest przy tym wykonywana żadna kontrola, więc użycie bloków kodu wymaga od programisty dobrej znajomości języka maszynowego.

Format bloku kodu jest następujący:

[‹wartość›|: ‹wartość›:|]

gdzie **‹wartość›** jest jedną z wartości zapisanych w bloku kodu. Musi to być stała kompilatora. Jeżeli jest ona większa od 255, to jest zapisywana w dwóch bajtach, w kolejności młodszy, starszy (LSB, MSB).

Przykłady:

[$40 $0D $51 $F0 $600]
BYTE b1,b2,b3
['A b1 342 b3 4+$A7]
DEFINE on=„1"
[54 on on+'t $FFF8]

Bloki kodu są wygodne przy umieszczaniu w programie krótkich procedur w języku maszynowym, lecz zbyt kłopotliwe przy dłuższych. Użycie dłuższych procedur lub większej ich liczby wymaga zastosowania innego sposobu (patrz dalej).

Adresowanie zmiennych

W rozdziałach poświęconych deklarowaniu zmiennych podstawowych, tablicowych i wskaźnikowych widziałeś, że adres zmiennej może być określony w deklaracji, lecz nie było tam przykładów wykorzystania tej możliwości.

```
PROC kolory()

  BYTE wsync=54282,
       vcount=54283,
       clr=53272,
       chgclr=[0],
       ctr,incclr
```

```
  Graphics(0)
  PutE()
  FOR ctr=1 TO 23
    DO
      PrintE("ZMIANA KOLOROW TLA")
    OD
  Print("ZMIANA KOLOROW TLA")
  DO
    FOR ctr=1 TO 4
      DO
        incclr=chgclr
        DO
          wsync=0
          clr=incclr
          incclr==+1
          UNTIL vcount&128
        OD
      OD
    chgclr==+1
  OD
RETURN
```

Pozwala to na deklarowanie zmiennych **Action!**, które mają takie same adresy jak rejestry sprzętowe. Można w ten sposób bezpośrednio operować grafiką i dźwiękiem, zmieniać parametry pracy systemu operacyjnego itd. Ilustracją tych możliwości będzie program, który zmienia i przesuwa kolory tła. W tym celu nie można użyć normalnych rejestrów koloru, gdyż wpływają one na wygląd obrazu tylko po każdym przerwaniu synchronizacji pionowej (co 1/50 sekundy). Zamiast tego należy bezpośrednio operować na sprzętowych rejestrach koloru. W ten sposób można zmieniać kolor tła w trakcie tworzenia każdego obrazu. W programie jest to realizowane 12 razy (otrzymujemy więc 12 kolorów w trybie 0). Trzeba przy tym zapewnić niezmienność koloru w obrębie linii ekranu, do czego służy rejestr sprzętowy WSYNC, który wskazuje, kiedy jedna linia ekranu została utworzona, a następna jeszcze nie została rozpoczęta. Zmienna VCOUNT pokazuje natomiast, jak dużo linii ekranu zostało już utworzonych i jest wykorzystana do regulowania przesuwu.

Adresowanie procedur

Idea określania adresu procedury jest podobna do określania adresu zmiennej. W poprzednim ustępie pokazaliśmy bezpośrednie użycie rejestrów systemowych Atari poprzez odpowiednie adresowanie zmiennych **Action!**. Ponieważ możesz zdefiniować adres procedury, to możesz bezpośrednio wywoływać procedury systemu operacyjnego oraz własne procedury w języku maszynowym. Użyta metoda jest opisana w następnym ustępie, gdyż dotyczy ona wszystkich procedur w języku maszynowym — zarówno napisanych przez Ciebie, jak i wbudowanych w systemie operacyjnym.

Procedury maszynowe

Action! pozwala na bardzo łatwe wywoływanie procedur maszynowych. Wymagane są do tego tylko dwie rzeczy:
— musisz znać adres początkowy procedury,
— procedura musi kończyć się rozkazem „RTS" (jeśli chcesz wrócić do **Action!**).

Dla osób programujących w języku maszynowym wymagania te nie są trudne do spełnienia.

„A co z parametrami?" Tak, istnieje możliwość przekazania parametrów do procedury w języku maszynowej. Kompilator przechowuje parametry w następujący sposób:

adres	kolejny bajt parametrów
akumulator	pierwszy
rejestr X	drugi
rejestr Y	trzeci
$A3	czwarty
$A4	piąty
:	:
$AF	szesnasty

A teraz przykład (UWAGA: W niektórych wersjach **Action!** podczas kompilacji może wystąpić błąd 14. Należy w takim przypadku zmniejszyć wartość buflen'.):

```
PROC CIO=$E456(BYTE aku,xreg)
;*** Deklaracja systemowej procedury
;CIO. 'xreg' zawiera numer kanalu
;pomnozony przez 16, a 'aku' jest
;liczba umieszczana w akumulatorze.

PROC odczyt2()
;*** Procedura otwiera kanal 2 i
;odczytuje z niego 'buflen' bajtow.

  DEFINE buflen="$2000"

  BYTE ARRAY fname(30),
             buffer(buflen)

  BYTE iocb2cmd=$362

  CARD iocb2buf=$364,
       iocb2len=$368

  PutE()
  Print("Nazwa pliku >>  ")
  InputS(fname)
  Open(2,fname,4,0)
  iocb2cmd=7
  iocb2buf=buffer
  iocb2len=buflen
  CIO(0,$20) ;**** wywolanie CIO ****
  Close(2)
RETURN
```

Widzisz, jakie to proste? Posiadaczom biblioteki procedur w języku maszynowym umożliwia to wykorzystanie ich w programie **Action!** dla uzyskania efektywniejszego programu i dla uproszczenia jego tworzenia.

Użycie parametrów

Omawialiśmy już uprzednio parametry i ich wykorzystanie. Stwierdziliśmy przy tym, że nie można przekazać wartości z procedury przy użyciu parametru. Nie do końca jest to prawdą. Można przekazać parametrem wartość z procedury, jeżeli wykorzysta się w tym celu wskaźnik. Trzeba tylko utworzyć wskaźnik wskazujący zmienną, którą rzeczywiście chcesz przekazać do procedury i zamiast niej przekazać ten wskaźnik. Gdy następnie wykorzystujesz miejsce wskazywane przez wskaźnik, w rzeczywistości jest to zmienna, którą chciałeś przekazać. Możesz teraz zmienić wartość tej zmiennej używając wskaźnika.

Metoda ta wymaga pewnego pośrednictwa (tzn. użycia wskaźnika zmiennej zamiast samej zmiennej), lecz jest to w niektórych przypadkach bardzo efektywne i użyteczne, jak pokazuje poniższy przykład.

```
BYTE FUNC substr(BYTE ARRAY str,sub
          BYTE POINTER errptr,notfnd)

;*** Funkcja poszukuje w ciagu 'str'
;podciagu 'sub'. Po znalezieniu
;zwraca indeks w ciagu. Jesli podciag
;jest dluzszy niz ciag glowny, to
;wskazywany jest blad. Jesli podciag
;nie zostanie znaleziony, informacja
;o tym jest przekazywana przez inny
;wskaznik.

  BYTE ARRAY tempstr

  BYTE ctr1,ctr2

  IF sub(0)>str(0) THEN
    errptr^=1
  ELSE
    FOR ctr1=1 TO str(0)
      DO
      IF sub(1)=str(ctr1) THEN
        tempstr(0)=sub(0)
        FOR ctr2=1 TO sub(0)
          DO
          tempstr(ctr2)=str(ctr2+ctr1-1)
          OD
        IF SCompare(tempstr,sub)=0 THEN
          RETURN(ctr1)
        FI
      FI
      OD
  FI
  notfnd^=1
RETURN(0)
```

Ten rodzaj operowania parametrami wymaga pewnej wprawy w operowaniu wskaźnikami. Jest to jednak najszybszy najłatwiejszy sposób przekazania z procedury większej liczby parametrów bez użycia zmiennych globalnych. Oznacza to, że procedura nadal może mieć wiele zastosowań, niezależnie od wykorzystywanych w programie zmiennych globalnych.

Wojciech Zientara

Opis języka **Action!** publikowany w „Moim Atari" był rozpoczęty w wydaniach specjalnych „Bajtka" — „Tylko o Atari". Zdajemy sobie sprawę z tego, że znaczna część czytelników nie ma dostępu do tych pism. Możemy powtórnie je wydrukować, jeżeli takie życzenie zostanie wyrażone w listach do redakcji. Proponujemy jednak nieco inną formę, a mianowicie publikowanie tych informacji „po kawałku". Na początek powtarzamy wykaz poleceń rozpoznawanych przez edytor **Action!.**

POLECENIA EDYTORA

przejście do monitora *(Monitor)* ‹CTRL›‹SHIFT›‹M›

Polecenia wejścia/wyjścia

odczyt pliku *(Read)* ‹CTRL›‹SHIFT›‹R› — nazwa
odczyt katalogu *(Read)* ‹CTRL›‹SHIFT›‹R› — ?n:*.*
zapis pliku *(Write)* ‹CTRL›‹SHIFT›‹W› — nazwa
wydruk pliku *(Write)* ‹CTRL›‹SHIFT›‹W› — P:

Ruch kursora

kursor na początek pliku *(Head)* ‹CTRL›‹SHIFT›‹H›
kursor na koniec pliku *(End)* ‹CTRL›‹SHIFT›‹E›
kursor o 1 wiersz w górę ‹CTRL›‹—›
kursor o 1 wiersz w dół ‹CTRL›‹=
kursor o 1 znak w prawo ‹CTRL›‹*›
kursor o 1 znak w lewo ‹CTRL›‹+›
kursor na początek wiersza ‹CTRL›‹SHIFT›‹CLEAR›
kursor na koniec wiersza ‹CTRL›‹SHIFT›‹INSERT›
następny wiersz ‹RETURN›
tabulacja ‹TAB›
ustawienie tabulacji ‹SHIFT›‹TAB›
skasowanie tabulacji ‹CTRL›‹TAB›

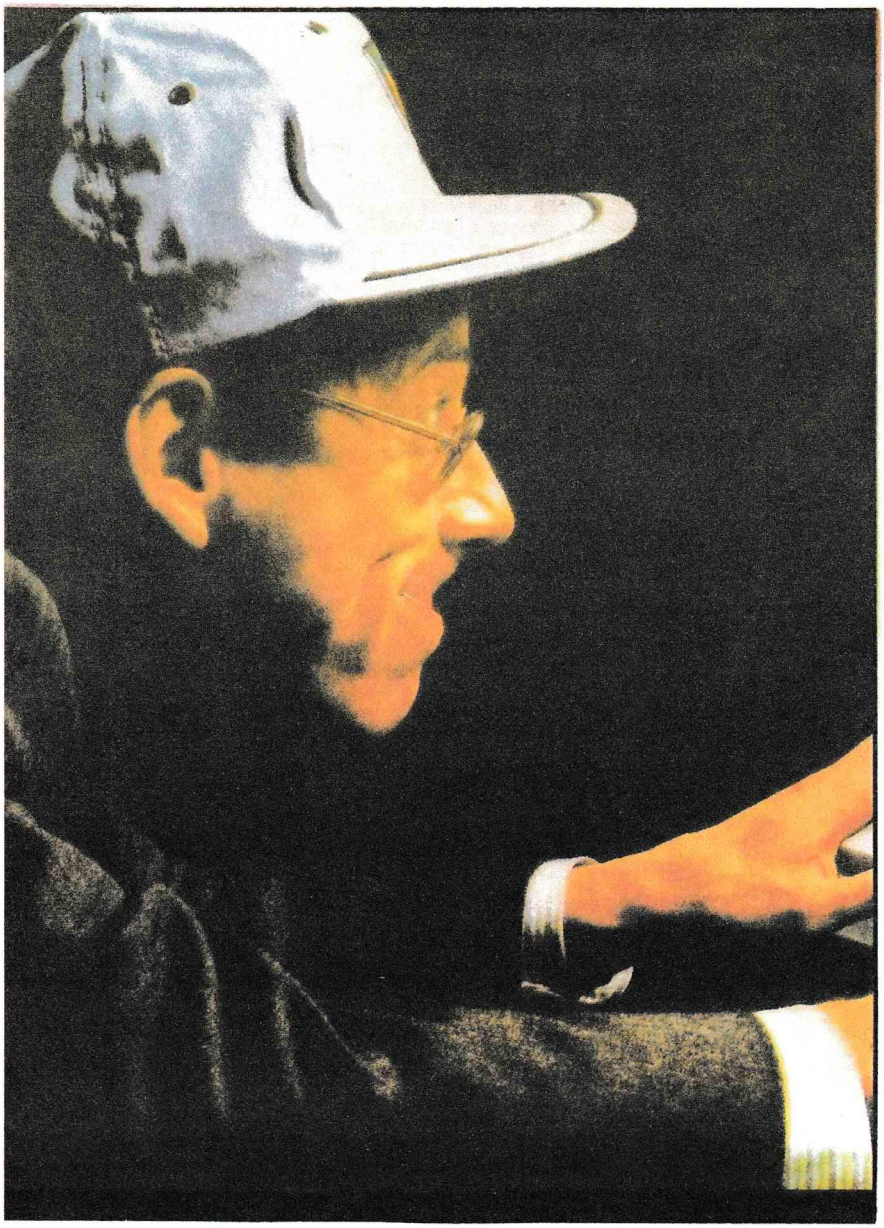
Ruch okna ekranu

okno o 1 ekran w górę ‹CTRL›‹SHIFT›‹—›
okno o 1 ekran w dół ‹CTRL›‹SHIFT›‹=›
okno o 1 znak w lewo ‹CTRL›‹SHIFT›‹]›
okno o 1 znak w prawo ‹CTRL›‹SHIFT›‹[›
utworzenie drugiego okna ‹CTRL›‹SHIFT›‹2›
przejście do pierwszego okna ‹CTRL›‹SHIFT›‹1›
przejście do drugiego okna ‹CTRL›‹SHIFT›‹2›
skasowanie okna *(Delete)* ‹CTRL›‹SHIFT›‹D›

Redagowanie tekstu

wstawianie/wymiana *(Insert/replace)* ‹CTRL›‹SHIFT›‹I›
odtworzenie zmienionego wiersza *(Undone)* ‹CTRL›‹SHIFT›‹U›
usunięcie wiersza ‹SHIFT›‹DELETE›
odtworzenie usuniętego wiersza *(Paste)* ‹CTRL›‹SHIFT›‹P›
zapamiętanie bloku tekstu w buforze ‹SHIFT›‹DELETE›
wstawienie tekstu z bufora *(Paste)* ‹CTRL›‹SHIFT›‹P›
wyszukiwanie ciągu znaków *(Find)* ‹CTRL›‹SHIFT›‹F›
wymiana ciągu znaków *(Substitute)* ‹CTRL›‹SHIFT›‹S›
podzielenie wiersza na dwa ‹CTRL›‹SHIFT›‹RETURN›
połączenie dwóch wierszy ‹CTRL›‹SHIFT›‹DELETE›
ustawienie etykiety *(Tag set)* ‹CTRL›‹SHIFT›‹T›
odszukanie etykiety *(Go to tag)* ‹CTRL›‹SHIFT›‹G›

Wojciech Zientara



ACTION! (6)

Ostatnia już część opisu języka Action! dotyczy działania kompilatora oraz zasad programowania i uruchamiania programów w tym języku. Na końcu zostały ponadto podane zasady składni Action! i konwersji programów z Atari Basic.

KOMPILATOR ACTION!

Atari Basic pozwala pisać program w języku zbliżonym do angielskiego, a następnie bezpośrednio sprawdzić jego działanie bez realizowania dodatkowych operacji. Zaleta ta jest okupiona koniecznością istnienia specjalnego programu (zwanego interpreterem Basica), który rozpoznaje i wykonuje każdą instrukcję w każdym wierszu podczas realizacji programu.

Action! jest nieco bardziej złożony. Wymaga przekształcenia twojego programu przed jego realizacją przez specjalny program (zwany kompilatorem). Konieczne jest w tym celu wykonanie dodatkowej operacji między wpisaniem programu a jego uruchomieniem. Operacja ta nazywa się kompilacją. Podczas kompilacji kompilator **Action!** analizuje twój program wiersz po wierszu i przekształca go na program w języku maszynowym komputera. Dopiero tak przekształcony (skompilowany) program może być uruchomiony i działa wtedy znacznie szybciej niż interpretowany program w Atari Basic.

W tej części opisu będziemy używać kilku terminów, omawianych już wcześniej. Są to:

<ident> dowolny poprawny identyfikator
<wart> dowolna wartość dziesiętna lub szesnastkowa
<st_komp> obliczony adres identyfikatora
<adres> adres komórki pamięci

Ponadto zajmiemy się ponownie — nieco dokładniej — dyrektywami kompilatora. Przypominam, że dyrektywy te są wykonywane podczas kompilacji programu, a nie podczas jego działania i nie można ich użyć do zmiany parametrów systemu po uruchomieniu programu.

WYKORZYSTANIE PAMIĘCI

Opiszę teraz, jak kompilator **Action!** wykorzystuje dostępny obszar pamięci komputera do umieszczenia w nim skompilowanego programu, zmiennych, procedur i tablic symboli.

Po wywołaniu kompilator najpierw określa miejsce umieszczenia kodu wynikowego, uzyskanego po komplikacji programu źródłowego w **Action!**. Wykonuje on to przez sprawdzenie zawartości komórek pamięci 14 i 15. Zapisana w nich liczba typu **CARD** jest adresem początku wolnej pamięci. Adres ten jest zmienny i zależy od rozmiaru programu zapisanego w edytorze. Jeżeli nie wskażesz inaczej, to skomplikowany program jest umieszczany w pamięci od tego adresu. Dla wskazania kompilatorowi, gdzie ma umieścić skompilowany program, należy bezpośrednio przed kompilacją podać monitorowi dwa następujące polecenia:

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

gdzie <adres> jest adresem początku skompilowanego programu.

Trzeba przy tym pamiętać, że komentarze, dyrektywy **SET** i dyrektywy **DEFINE** nie tworzą żadnego kodu wynikowego. Jest tak, ponieważ nie działają one podczas pracy programu, a więc ich kod wynikowy jest zbędny.

Umieszczenie zmiennych

Informacje o zmiennych są przez kompilator **Action!** umieszczane w dwóch różnych miejscach — wewnątrz samego programu wynikowego i w tablicy symboli. Tablica ta jest opisana nieco dalej.

Wewnątrz programu zmienne są umieszczane przed tą częścią kodu maszynowego, w której zostały użyte. Niektóre zmienne są deklarowane przed pierwszą procedurą. Zmienne te (zwane globalnymi) mogą być użyte przez każdą znajdującą się dalej procedurę. Nie wymagają one dodatkowych deklaracji wewnątrz procedury.

Miejsce przeznaczane na zapisanie zmiennych zależy od ich typu. Poniższa tabela pomoże ci zrozumieć, w jaki sposób kompilator umieszcza zmienne.

typ danej	miejsce	komentarz
BYTE	1 bajt	typ podstawowy
CHAR	1 bajt	typ podstawowy
CARD	2 bajty	typ podstawowy
INT	2 bajty	typ podstawowy
ARRAY	rozmiar typu razy liczba elementów	typ rozszerzony
TYPE	suma rozmiarów typów podanych w deklaracji	typ rozszerzony
ciąg	liczba znaków w ciągu plus bajt określający długość	każdy ciąg jest zapisywany oddzielnie nawet, jeśli ma taki sam identyfikator

Procedury i funkcje

Kompilator przeznacza dla procedur i funkcji miejsce w pamięci za zmiennymi globalnymi. Zmienne deklarowane lokalnie w procedurze poprzedzają rozkazy kodu wynikowego tej procedury. Tekst programu (instrukcje wewnątrz procedury lub funkcji) jest zamieniany bezpośrednio na rozkazy kodu maszynowego.

Programy dołączone

W dowolnym miejscu programu można do niego dołączyć inne programy przy użyciu dyrektywy **INCLUDE**. Oczywiście, dołączany tekst nie może kolidować z aktualnie kompilowanym programem. Chodzi tu przede wszystkim o zbieżność identyfikatorów. Gdy w dołączonym programie zostanie wykryty błąd, jest on zwykle wyświetlany w obszarze ekranu przeznaczonym na komunikaty. Numer błędu zawsze jest pokazywany w wierszu poleceń monitora i sygnalizowany dźwiękiem.

Dodatkowe zmienne globalne

Dodatkowe globalne zmienne, tablice i rekordy mogą być dodawane w razie konieczności przez użycie słowa kluczowego **MODULE**. Zmienne te zajmują miejsce za kodem wynikowym ostatniej poprzedzającej je procedury. Ich identyfikatory są dołączane do globalnej tablicy symboli kompilatora i mogą być wykorzystywane przez wszystkie następujące po nich procedury i funkcje.

Tablice symboli

Kompilator **Action!** przechowuje dwie tablice symboli — jedną dla zmiennych globalnych i jedną dla zmiennych lokalnych z ostatnio kompilowanej procedury. Tablice symboli są dostępne dla użytkownika z monitora **Action!** poprzez polecenia **?**, * i **SET** (patrz opis monitora w „Tylko o Atari 1"). Mogą one być również użyte przez kompilator, gdy wymagany jest adres zmiennej.

Kompilator przeznacza na te tablice osiem stron pamięci (2 KB) znajdujące się na szczycie dostępnej pamięci. Ponieważ w ten sposób znajdują się one bezpośrednio poniżej pamięci obrazu, to są niszczone przy zmianie trybu graficznego na taki, który wymaga więcej miejsca niż tryby 0. Oznacza to, że nie możesz wrócić do monitora podczas wykonywania programu w celu sprawdzenia wartości zmiennych. Nie ma to jednak żadnego znaczenia dla poprawnego działania programu.

UŻYCIE MENU WARIANTÓW

Menu wariantów oferuje szereg sposobów rozszerzenia lub zmiany działania kompilatora **Action!**. Warianty te były już częściowo opisane poprzednio.

Zwiększenie szybkości kompilatora:

Możesz osiągnąć zwiększenie szybkości kompilacji maksymalnie o 30% przez wykorzystanie menu wariantów do wyłączenia obrazu podczas dyskowych operacji wejścia/wyjścia i kompilacji programu. Po prostu wpisz „N<RETURN>" w odpowiedzi na pytanie „Screen?" wyświetlone w menu wariantów.

UWAGA: Wyłącza to również obraz dla innych funkcji systemu **Action!**, więc po zakończeniu kompilacji powinieneś ponownie przywrócić wyświetlanie obrazu.

Wyłączenie brzęczyka:

Gdy testujesz nowy program zawierający wiele błędów, możesz wyłączyć dźwięk brzęczyka. Wpisz tylko „N<RETURN>" w odpowiedzi na „Bell?" w menu wariantów.

Rozpoznawanie wielkości liter

Niekiedy, szczególnie gdy jesteś początkującym programistą, możesz żądać, aby kompilator pomagał ci w programowaniu przez wskazywanie słów kluczowych **Action!** wpisanych małymi literami. Możesz także chcieć, aby duże i małe litery w identyfikatorach były rozróżniane. Uzyskasz te funkcje wpisując w menu wariantów „Y<RETURN>" w odpowiedzi na pytanie „Case sensitive?".

Wykorzystanie tego wariantu nie jest konieczne do poprawnego programowania w **Action!**. Jednakże jest on użyteczny, gdy stosujesz dużą liczbę różnych identyfikatorów.

Listing kompilacji

Możesz nakazać kompilatorowi wyświetlanie każdego wiersza programu, jaki jest komplikowany. Nie jest to konieczne, gdyż większość występujących błędów jest wyświetlana na ekranie podczas procesu kompilacji. Możesz jednak kompilować długi program, który zawiera procedury dołączane (**INCLUDE**) z innych plików źródłowych. Jeśli tak, to nigdy nie możesz obejrzeć w inny sposób pełnego listingu programu źródłowego. Możesz go otrzymać na ekranie, a nawet wydrukować (przez zmianę

TABELA 1

Stałe Action!

Stałe liczbowe

```
<st_liczb> ::= <liczba_dec> | <liczba_hex> | <znak>
<liczba_dec> ::= <liczba_dec><cyfra> | <cyfra>
<liczba_hex> ::= <liczba_hex><cyfra_hex> | $<cyfra_hex>
<znak> ::= '<dowolny drukowalny znak>
<cyfra_hex> ::= <cyfra> | A | B | C | D | E | F
<cyfra> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Stałe tekstowe

```
<st_tekst> ::= "<ciag>"
<ciag> ::= <ciag><znak_tekst> | <znak_tekst>
<znak_tekst> ::= <kazdy drukowalny znak oprocz ">
```

Stałe kompilatora

```
<st_komp> ::= <st_komp>+<podst_st_komp> | <podst_st_komp>
<podst_st_komp> ::= <ident> | <st_liczb> | <odw_wsk> | *
```

Operatory i podstawowe typy danych

Operatory

```
<op_spec> ::= AND | OR | & | %
<op_rel> ::= XOR | ! | = | # | <> | < | <= | > | >=
<op_dod> ::= + | -
<op_mnoż> ::= * | / | MOD | LSH | RSH
<op_larg> ::= @ | -
```

Podstawowe typy danych

```
<typ_podst> ::= CARD | CHAR | BYTE | INT
```

Struktura programu

Program w Action!

```
<program> ::= <program> MODULE <mod_prog> | {MODULE} <mod_prog>
<mod_prog> ::| {<dekl_syst>} <lista_proc>
```

Deklaracje systemowe

```
<dekl_syst> ::= <dyr_DEFINE> | <dekl_TYPE> | <dekl_zm>
```

Dyrektywa DEFINE

```
<dyr_DEFINE> ::= <DEFINE><lista_def>
<lista_def> ::= <lista_def>,<def> | <def>
<def> ::= <ident>=<st_tekst>
```

Deklaracja TYPE

```
<dekl_TYPE> ::= TYPE <lista_id_rek> <id_rek> | <id_rek>
<id_rek> ::= <nazwa_rek>=[<pole_pocz>]
<nazwa_rek> ::= <ident>
<pole_pocz> ::= <dekl_podst>
```

Deklaracja zmiennej

```
<dekl_zm> ::= <dekl_zm> <podst_dekl_zm> | <podst_dekl_zm>
<podst_dekl_zm> ::= <dekl_podst> | <dekl_wsk> | <dekl_tabl> | <dekl_rek>
```

Deklaracja zmiennej typu podstawowego

```
<dekl_podst> ::= <dekl_podst> <dekl_prosta> | <dekl_prosta>
<dekl_prosta> ::= <typ_podst> <lista_id_podst>
<typ_podst> ::= CARD | CHAR | BYTE | INT
<lista_id_podst> ::= <lista_id_podst>,<id_podst> | <id_podst>
<id_podst> ::= <ident>{=<war_pocz>}
<war_pocz> ::= <adres> | [<wartość>]
<adres> ::= <st_komp>
<wartość>::= <st_liczb>
```

Deklaracja zmiennej wskaźnikowej

```
<dekl_wsk> ::= <typ_wsk> POINTER <lista_id_wsk>
<typ_wsk> ::= <typ_podst> | <nazwa_rek>
<lista_id_wsk> ::= <lista_id_wsk>,<id_wsk> | <id_wsk>
<id_wsk> ::= <ident>{=<wartość>}
```

Deklaracja zmiennej tablicowej

```
<dekl_tab> ::= <typ_podst> ARRAY <lista_id_tab>
<lista_id_tab> ::= <lista_id_tab>,<id_tab> | <id_tab>
<id_tab> ::= <ident>{(<roza>)}{=<war_pocz_tab>}
<rozm> ::= <st_liczb>
<war_pocz_tab> ::= <adres> | [<lista_wart>] | <st_tekst>
<adres> ::= <st_komp>
<lista_war> ::= <lista_war><wartość> | <wartość>
<wartość> ::= <st_komp>
```

wyjścia), gdy odpowiesz „Y<RETURN>" na pytanie „List?" w menu wariantów.

INFORMACJE TECHNICZNE

Przepełnienie

Kompilator **Action!** nie sprawdza przepełnienia arytmetycznego (ang. *overflow* lub *underflow*). A co to jest przepełnienie?

Jeżeli masz zmienną typu BYTE o wartości 255 i dodasz do niej 1, to otrzymasz 0, a nie 256 (ponieważ pojedynczy bajt może zawierać tylko wartość do 255). Podobnie, jeśli używasz systemu dziesiętnego i możesz wyświetlić tylko dwie cyfry, to natrafisz na taki sam problem dodając 1 do 99. Wiesz, że jest to 100, lecz możesz wyświetlić tylko dwie cyfry, więc zobaczysz „00". Jest to właśnie przepełnienie (*overflow*).

Przepełnienie ujemne (*underflow*) jest dokładnie odwrotne. Jeżeli odejmiesz 1 od zera, to otrzymasz 255.

Jak to było wcześniej opisane, niektóre operatory matematyczne dają określone typy wartości. Możesz więc czasem uniknąć tego problemu przez wykorzystanie automatycznej zmiany typu wartości.

Podobnie, przepełnienie jest powodowane przez operatory przesunięcia (**LSH** i **RSH**). Przesunięcie zawartości zmiennej. daje wynik podobny (lecz nie identyczny) jak mnożenie lub dzielenie przez 2.

Zgodność typów

Musisz również uważać, gdyż kompilator **Action!** nie sprawdza granic zmiennych prostych i tablic. Pominięcie tej kontroli pozwala na większą elastyczność w operowaniu danymi. Ceną za to jest konieczność zwiększenia uwagi. Możesz w tym celu napisać własne procedury kontrolujące granice tablic i wyświetlające komunikaty błędów. Procedury takie najlepiej wykorzystywać wielokrotnie w różnych programach przy pomocy dyrektywy **INCLUDE**.

Korzystanie z kanału 7

Po uruchomieniu systemu **Action!** kanał 7 jest otwierany do odczytu z klawiatury (K:). Możesz wykorzystywać ten kanał do tego samego celu, lecz nie zmieniaj jego atrybutów przez zamykanie go i ponowne otwieranie.

UWAGA: Jeżeli używasz kanału 7 i zakładasz, że jest on otwarty, to twój program nie będzie działał bez modułu **Action!**.

Dostępna pamięć

Przy pisaniu długiego programu wielkość dostępnej pamięci może okazać się niewystarczająca. Gdy to nastąpi, możesz zrobić jedną z trzech rzeczy, zależnie od chwili wystąpienia błędu.

Podczas redagowania programu

Zapisz tworzony program na kasecie lub dyskietce (<CTRL><SHIFT>W), przejdź do monitora i ponownie uruchom system (**BOOT**). Następnie wróć do edytora i odczytaj zapisany uprzednio program.

Jeżeli to nie pomaga (program jest bardzo długi), to trzeba podzielić go na mniejsze części i połączyć przy pomocy dyrektywy **INCLUDE**.

Podczas kompilacji programu

Przejdź do edytora i zapisz program. Następnie wróć do monitora, uruchom ponownie system i kompiluj program z nośnika (kasety lub dyskietki), na którym został zapisany. Jeżeli nie da to rezultatu, konieczne jest skrócenie programu lub zmiana jego wersji (patrz niżej).

Wersje Action!

Oryginalny język **Action!** jest sprzedawany w postaci modułu ROM (*cartridge*). Ponadto dostępne są piracki (kradzione) wersje na kasetach i dyskietkach. Cały system **Action!** zajmuje 16 KB pamięci, lecz w module ROM jest on podzielony na części włączane w razie potrzeby i w rzeczywistości zajmuje tylko 8 KB. Wersje pirackie nie mają tej możliwości i w związku z tym pozostawiają użytkownikowi o 8 KB, czyli o 1/3 pamięci mniej. Ponadto działają one znacznie wolniej podczas redagowania i kompilacji programu.

Ze względu na procedury i funkcje biblioteczne skompilowany program wymaga obecności **Action!** w pamięci komputera. Ominięcie tego warunku jest możliwe trzema różnymi sposobami. Po pierwsze, firma **ICD** sprzedaje wersję języka, która do skompilowanego programu dołącza procedury biblioteczne. Nie słyszałem jednak, aby ktokolwiek w Polsce posiadał tą wersję. Po drugie, istnieją programy umożliwiające dołączenie procedur do kompilowanego programu. Zwykle są to procedury „wydobyte" z systemu (ten sposób jest najczęściej stosowany). Po trzecie, można napisać program, który nie będzie zawierał biblioteki procedur i mimo to będzie działał poprawnie nawet bez **Action!**. Wymaga to wprowadzenia wielu ograniczeń, przede wszystkim nie wolno w takim programie stosować procedur bibliotecznych.

Efektywne programowanie

Program w **Action!** jest w procesie kompilacji zamieniany na kod maszynowy w sposób bezpośredni, czyli tak, jak został zapisany przez użytkownika. W celu uzyskania szybkiego i zwartego programu wynikowego należy więc stosować w programie konstrukcje przeznaczone specjalnie dla **Action!** unikać zaś struktur przeniesionych z Basica. Jaskrawym przykładem są tu procedury biblioteczne **Poke** i **Peek**. Ich wygląd po kompilacji jest następujący:

Poke (752,1)	key=Peek(764)
LDA #240	LDA #252
LDX #2	LDX #2
LDY #1	JSR PEEK
JSR POKE	STA KEY

W rezultacie kod wynikowy w pierwszym wypadku ma długość dziewięciu bajtów, a w drugim — dziesięciu. Definiując komórki pamięci jako zmienne o określonym adresie uzyskujemy znacz-

TABELA 2

```
      Deklaracja zmiennej rekordowej
<dekl_rek> ::= <ident> <lista_id_rek>
<lista_id_rek> ::= <lista_id_rek>,<id_rek> | <id_rek>
<id_rek> ::= <ident>[=<adres>]
<adres> ::= <st_komp>

          Odwołania do zmiennych
<odw_pam> ::= <zaw_pam> | @<ident>
<zaw_pam> ::= <odw_podst> | <odw_tab> | <odw_wsk> | <odw_rek>
<odw_podst> ::= <ident>
<odw_tab> ::= <ident>(<wyr_arytm>)
<odw_wsk> ::= <ident>^
<odw_rek> ::= <ident>.<ident>

          Procedury Action!

      Procedury
<lista_proc> ::= <lista_proc> <proc> | <proc>
<proc> ::= <procedura> | <funkcja>

      Struktura procedury
<procedura> ::= <dekl_proc> [<dekl_syst>] [<lista_instr>] [RETURN]
<dekl_proc> ::= PROC <ident>[=<adres>][(<dekl_par>)]
<adres> ::= <st_komp>

      Struktura funkcji
<funkcja> ::= <dekl_funkc> [<dekl_syst>] [<lista_instr>] [RETURN(<wyr_arytm>)]
<dekl_funkc> ::= <typ_podst> FUNC <ident>[=<adres>][(<dekl_par>)]
<adres> ::= <st_komp>

      Wywołanie procedury
<wyw_proc> ::= <wyw_FUNC> | <wyw_PROC>
<wyw_FUNC> ::= <ident>[(<param>)]
<wyw_PROC> ::= <ident>[(<param>)]
      Parametry
<dekl_par> ::= <dekl_zm>

          Instrukcje
<lista_instr> ::= <lista_instr> <instr> | <instr>
<instr> ::= <instr_prosta> | <instr_strukt> | <blok_kodu>
<instr_prosta> ::= <instr_przyp> | <instr_EXIT> | <wyw_proc>
<instr_strukt> ::= <instr_IF> | <pętla_DO> | <pętla_WHILE> | <pętla_FOR>

      Instrukcja przypisania
<instr_przyp> ::= <zaw_pam>=<wyr_arytm>

      Instrukcja EXIT
<instr_EXIT> ::= EXIT

      Instrukcja IF
<instr_IF> ::= IF <wyr_war> THEN [<lista_instr>] [!:<rozsz_ELSEIF>:!]
                 [<rozsz_ELSE>] FI
<rozsz_ELSEIF> ::= ELSEIF <wyr_war> THEN [<lista_instr>]
<rozsz_ELSE> ::= ELSE [<lista_instr>]
<wyr_war> ::= <rel_złoż>

      Pętla DO-OD
<pętla_DO> ::= DO [<lista_instr>] [<instr_UNTIL>] OD
<instr_UNTIL> ::= UNTIL <wyr_war>

      Pętla WHILE
<pętla_WHILE> ::= WHILE <wyr_war> <pętla_DO>

      Pętla FOR
<pętla_FOR> ::= FOR <ident>=<start> TO <stop> [STEP <krok>] <pętla_DO>
<start> ::= <wyr_arytm>
<stop> ::= <wyr_arytm>
<krok> ::= <wyr_arytm>

      Blok kodu
<blok_kodu> ::= [<lista_st-komp>]
<lista_st_komp> ::= <lista_st_komp> <st_komp> | <st_komp>

          Wyrażenia

      Wyrażenia relacji
<rel_złoż> ::= <rel_złoż><op_spec><rel_prosta> | <rel_prosta><op_spec><rel_prosta>
<rel_prosta> ::= <wyr_arytm><op_rel><wyr_arytm>

      Wyrażenia arytmetyczne
<wyr_arytm> ::= <wyr_arytm><op_dod><wyr_mult> | <wyr_mult>
<wyr_mult> ::= <wyr_mult><op_mnoż><wartość> | <wartość>
<wartość> ::= <st_liczb> | <odw_pam> | <wyr_arytm>
```

TABELA 2

Instrukcje Basica	Odpowiedniki w Action!
C=D+I*A	c=d+i*a
IF A<>0 THEN B=1	IF a<>0 THEN b=1 FI
10 IF A=0 THEN 30 20 B=1:C=A*2 30 REM	IF a<>0 THEN b=1 c=a*2 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 ... NEXT I	FOR i=1 TO 100 DO ... OD
PRINT "WITAJ"	PrintE("WITAJ")
PRINT "WITAJ";	Print("WITAJ")
PRINT #5;"WITAJ"	PrintDE(5,"WITAJ")
PRINT #5;"WITAJ";	PrintD(5,"WITAJ")
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()
INPUT B$	Put('?) InputS(ba)
PUT #0,65	Put('A) lub Put(65) lub Put($41)
GET #C,B	b=GetD(c)
OPEN #1,4,0,"K:"	Open(1,"K:",4,0)
CLOSE #3	Close(3)
NOTE #1,C,B	Note(1,@c,@b)
POINT #1,C,B	Point(1,c,b)
XIO 18,#6,0,0,"S:"	XIO(6,0,18,0,0,"S:") albo procedura Fill()
B=PEEK(C)	b=Peek(c) lub lepiej ba=c b=ba^
POKE C,B	Poke(c,b) lub lepiej ba=c ba^=b
GRAPHICS 8	Graphics(8)
COLOR 3	color=3
DRAWTO C,D	DrawTo(c,d)
LOCATE C,D,B	b=Locate(c,d)
PLOT C,D	Plot(c,d)
POSITION C,D	Position(c,d)
SETCOLOR 0,1,C	SetColor(0,1,c)
GRAPHICS 24:COLOR C: PLOT 200,150:DRAWTO 120,20: POSITION 40,150:POKE 765,C: XIO 18,#6,0,0,"S:"	Graphics(24) color=c Plot(200,150) DrawTo(120,20) Fill(40,150)
SUUND 0,121,10,6	Sound(0,121,10,6)
C=PADDLE(B) C=PTRIG(B) C=STICK(B) C=STRIG(B)	c=Paddle(b) c=Ptrig(b) c=Stick(b) c=Strig(b)
B$=S$	SCopy(ba,s)
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$=STR$(I)	StrI(i,ba)
I=VAL(S$)	i=ValI(s)

ną oszczędność miejsca (definicja występuje tylko raz i nie daje ani jednego bajtu kodu wynikowego). Jeżeli więc pokazane wyżej rejestry zdefiniujemy jako:

```
BYTE cinh=752, kb=764
```

to odpowiedni fragment programu będzie wyglądał następująco:

```
cinh=1          key=kb
  LDA #1        LDA KB
  STA CINH      STA KEY
```

Długość kodu wynikowego wynosi teraz w pierwszym przypadku 4 bajty, a w drugim 5, czyli jest on dwukrotnie krótszy.

SKŁADNIA ACTION!

W tabeli 1 podana jest składnia języka **Action!** w notacji Backusa-Naura. W notacji tej występuje pięć symboli o specjalnym znaczeniu:

symbol	znaczenie
::=	jest określone jako
|	lub
{}	wariantowo
[: :]	wielokrotnie
‹ident›	identyfikator (nazwa)

Konwersja programów z Basica na Action!

Tabela 2 prezentuje liczne funkcje procedury i instrukcje Basica. Dla każdego przykładu w Basicu podana jest odpowiednia struktura w **Action!**. W przykładach w Basicu nie są podane numery wierszy, chyba że są one niezbędne dla zrozumienia przykładu. W poniższych przykładach w **Action!** zakładamy, że zostały wcześniej zadeklarowane następujące zmienne:

```
INT i,j,k
CARD c,d,e
BYTE a,b
BYTE ARRAY s,t,aa,ba
CARD ARRAY ca,da,ea
INT ARRAY ia,ja,ka
```

MAPA PAMIĘCI ACTION!

Poniższa mapa pamięci systemu **Action!** dotyczy wyłącznie wersji oryginalnej, umieszczonej w modelu ROM (taleba 3).

UWAGA: Obszar kodu kompilatora rozpoczyna się w miejscu, w którym kończy się bufor edytora. Czyni to dynamicznym położenie w pamięci zarówno edytora, jak i kompilatora.

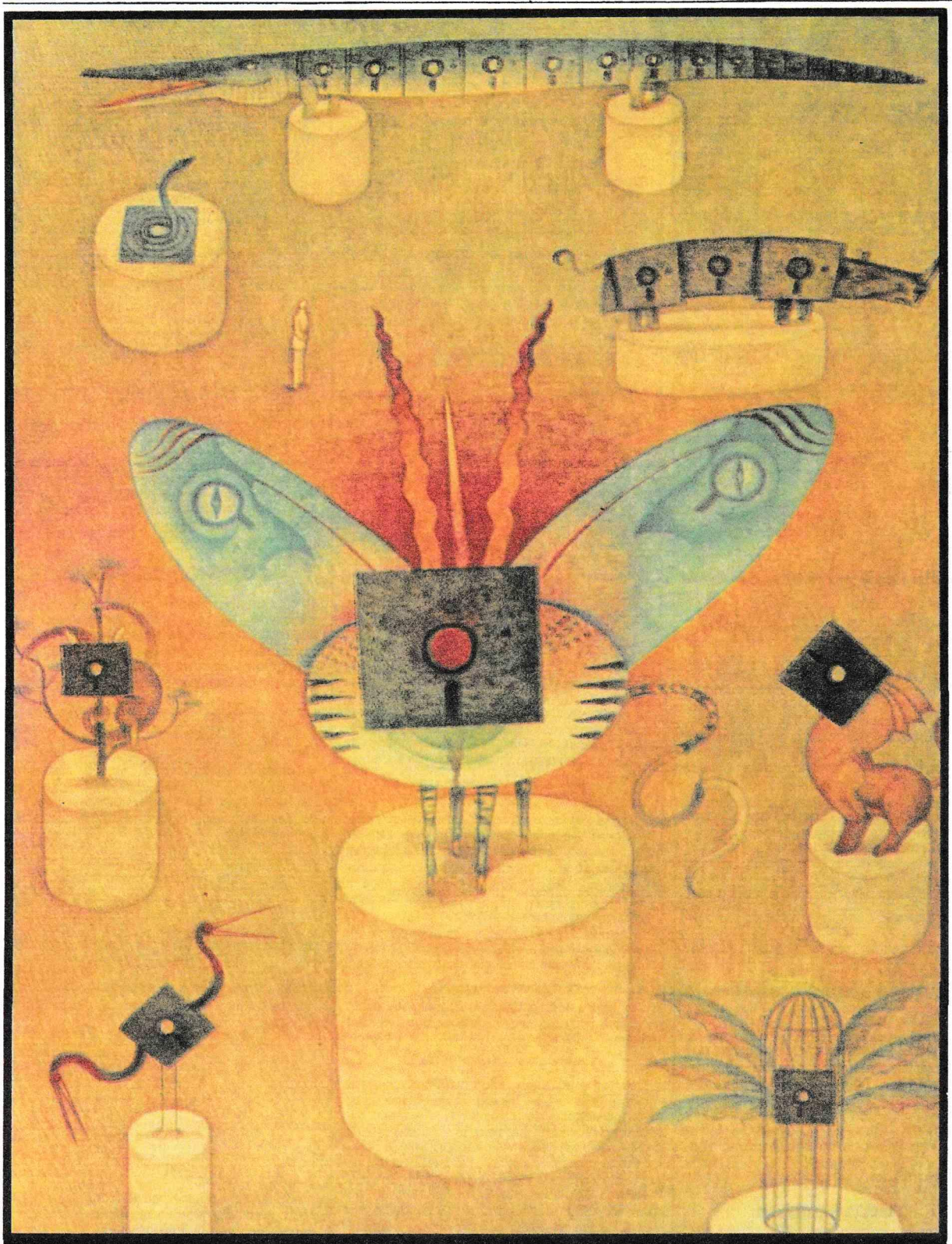
Wojciech Zientara

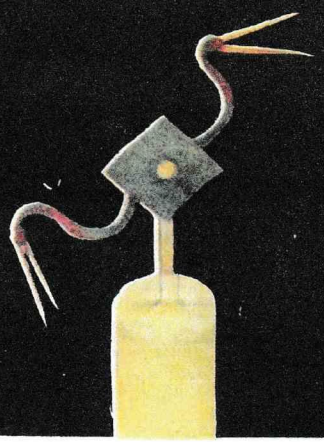
TABELA 3

$00	Zmienne OS i Action!
$CA	Wolny obszar
$CE	Zmienne Action!
$D4	Rejestry liczb rzeczywistych
$100	System Operacyjny
$480	Zmienne Action!
$580	Bufor liczb rzeczywistych
$600	System operacyjny
MEMLO	Stos kompilatora Action!
LO+$200	Bufor wejściowy Action!
LO+$300	Tablica danych Action!
LO+$750	Bufor edytora Action!
	Obszar kodu kompilatora Action!
TOP-$800	Tablica symboli kompilatora
MEMTOP	Pamięć obrazu
$A000	Moduł Action!
$C000	System operacyjny
$FFFF	

PROGRAMOWANIE

ACTION! BEZ ACTION!





Wielokrotnie — zarówno w opisie Action!, jak i w różnych artykułach w „Bajtku" — pisałem, że program napisany w Action! i skompilowany nie działa, jeżeli sam Action! nie znajduje się w pamięci komputera. Wynika to z faktu korzystania przez skompilowany program z procedur, które znajdują się w module języka. Wspominałem jednak, iż możliwe jest uruchomienie programu bez języka, jeśli tylko spełnionych zostanie kilka warunków.

Język Action! powstał w firmie OSS (obecnie jest ona połączona z ICD). Dla programistów, którzy chcą sprzedawać swoje programy, został również napisany specjalny program o nazwie „OSS Runtime Library" (biblioteka uruchamiająca). Po jego dołączeniu (za pomocą dyrektywy INCLUDE) otrzymamy w wyniku kompilacji program, który działa nawet wtedy, gdy Action! nie został uprzednio umieszczony w pamięci. Posiadacz takiego programu ma kłopot „z głowy".

PROGRAMOWANIE

Istnieje jednak możliwość uruchomienia programu w **Action!** bez biblioteki procedur. W tym celu trzeba podczas pisania programu przestrzegać czterech następujących zasad:

a) Nie wolno wykorzystywać ŻADNYCH procedur i funkcji bibliotecznych **Action!**.

b) Jako parametry do zdefiniowanych w programie procedur można przekazywać nie więcej niż 3 bajty. Trzeba przy tym pamiętać, że jedna zmienna typu CARD lub INT zawiera już dwa bajty (zmienna typu BYTE ma oczywiście jeden bajt). Jeżeli potrzeba więcej parametrów, to muszą być one zdefiniowane poza procedurą — jako zmienne globalne.

c) Nie wolno używać mnożenia ani dzielenia.

d) Operatory przesunięcia (RSH i LSH) mogą być stosowane tylko z argumentami typu BYTE. Nie wolno wykonywać operacji przesunięcia na zmiennych typu CARD i INT.

Program napisany zgodnie z powyższymi zasadami po skompilowaniu będzie działał niezależnie od obecności języka w pamięci komputera. Bardzo proste, prawda? Ale jak będzie wyglądał program bez GRAPHICS, PRINT itd.? Czy nie ma innego sposobu?

BIBLIOTEKA MODUŁU

Jak wiesz (lub też nie), obszar pamięci zajmowany przez moduł **Action!** rozciąga się od adresu $A000 do $C000. Znajdują się tam również procedury w języku maszynowym realizujące procedury i funkcje **Action!**, jak np. PRINT, INPUT. Najprościej byłoby zapisać zawartość tego obszaru pamięci na dyskietce (funkcja DOS-u "K": ACTION.LIB,A000,BFFF). Naturalnie można tak zrobić. Pojawia się jednak następujący problem: obszar pamięci A000—C000 nie jest rezerwowany przy uruchamianiu komputera z wciśniętym klawiszem ‹OPTION› (znajduje się tam między innymi pamięć obrazu). Dlatego też przed odczytem procedur **Action!** trzeba zmienić w komórce 106 (numer najwyższej będącej do dyspozycji strony pamięci) zawartość ze 192 na 160 i otworzyć ponownie kanał graficzny (np. kanał 0). Jest to wystarczająco skomplikowane, aby zniechęcić przeciętnego użytkownika.

WŁASNA BIBLIOTEKA

Czy nie ma z tej sytuacji wyjścia? Przypomnij sobie, że zdefiniowanie procedury, której nazwa była już wcześniej użyta, powoduje skasowanie poprzedniej definicji. Na przykład zdefiniowanie procedury **Open()** spowoduje, że wszystkie odwołania będą wywoływały naszą procedurę, zamiast procedury bibliotecznej znajdującej się w **Action!**. Wystarczy więc zdefiniować własną bibliotekę procedur **Action!** i zapisać ją pod nazwą "LIB.ACT". Teraz na początku tworzonego programu piszemy:

INCLUDE "LIB.ACT"

a następnie:

MODULE

po czym umieszczamy definicje zmiennych globalnych występujących w programie oraz jego procedury i funkcje.

Naturalnie można również plik LIB.ACT wczytać bezpośrednio do edytora **Action!** i dopisać do niego pozostałą część programu.

UWAGA: Plik "LIB.ACT" znajdujący się na dyskietce „Moje Atari 5" nie jest gotową biblioteką, lecz tylko zbiorem zamieszczonych poniżej procedur.

Proste procedury i funkcje

Najprostsze do zastąpienia są (jak zwykle) te procedury, które są rzadko używane lub nawet zbędne. Należą do nich przede wszystkim funkcje manipulatorów, których działanie polega na odczytywaniu zawartości określonych komórek pamięci. Można je zastąpić zestawem następujących funkcji:

```
; funkcje manipulatorow

BYTE FUNC Stick(BYTE port)
 BYTE ARRAY joy(2)=$278
RETURN(joy(port))

BYTE FUNC Strig(BYTE port)
 BYTE ARRAY trig(2)=$284
RETURN(trig(port))

BYTE FUNC Paddle(BYTE port)
 BYTE ARRAY pad(4)=$270
RETURN(pad(port))

BYTE FUNC Ptrig(BYTE port)
 BYTE ARRAY ptrg(4)=$27C
RETURN(ptrg(port))
```

Jest to jednak zupełnie zbędne, gdyż zamiast tego wystarczy na początku programu zdefiniować cztery tablice:

```
BYTE ARRAY joy(2)=$278,trig(2)=$284,
            pad(4)=$270,ptrg(4)=$27C
```

Równie łatwa jest do zdefiniowania procedura **Position()**, która podane wartości umieszcza w odpowiednich rejestrach. Niejako przy okazji można także zdefiniować dodatkową procedurę **TextPosition()**, która umożliwi sterowanie położeniem kursora w oknie tekstowym, a której oryginalny **Action!** nie posiada.

```
PROC Position(CARD col BYTE row)
 BYTE crsrow=$54
 CARD crscol=$55
  crsrow=row
  crscol=col
RETURN

PROC TextPosition(CARD col BYTE row)
 BYTE txtrow=$290
 CARD txtcol=$291
  txtrow=row
  txtcol=col
RETURN
```

Zamiast wywoływać takie procedury można wszakże bezpośrednio umieszczać żądane wartości w tych rejestrach. Dodatkową zaletą takiego rozwiązania jest możliwość zmieniania tylko jednej współrzędnej kursora. W tym celu trzeba na początku zdefiniować zmienne:

```
BYTE crsrow=$54,txtrow=$290
CARD crscol=$55,txtcol=$291
```

Zbliżona jest także konstrukcja służącej do zmiany kolorów procedury **SetColor()**, która zamiast odrębnych zmiennych zawiera pięcioelementową tablicę:

```
PROC SetColor(BYTE reg,hue,lum)
 BYTE ARRAY col(5)=$2C4
  col(reg)=hue LSH 4 + lum
RETURN
```

Tablica ta może być zdefiniowana na początku programu jako zmienna globalna:

```
BYTE ARRAY col(5)=$2C4
```

Wybór koloru w programie następuje wtedy przez wykonanie instrukcji przypisania, w której zamiast zmiennych można umieścić również stałe wartości:

```
col(reg)=hue LSH 4 + lum
```

Ostatnią w tej grupie, choć nieco trudniejszą w realizacji, jest funkcja **Rand()**, która zwraca liczbę losową o wartości nie przekraczającej wartości parametrem procedury jest zero, to zwracana jest po prostu zawartość tego rejestru. Dla innych parametrem procedury jest zero, to zwracana jest po po prostu zawartość tego rejestru. Dla innych parametrów zwracana wartość losowa musi być resztą z dzielenia zawartości rejestru przez parametr. Ponieważ dzielenie jest niedozwolone, zastosujmy odejmowanie w pętli WHILE.

```
; procedura losowa

BYTE FUNC Rand(BYTE range)
 BYTE rnd=$D20A,i
  IF range=0 THEN
    RETURN(rnd)
  ELSE
    i=rnd
  FI
  WHILE i>=range
   DO
    i=i-range
   OD
RETURN(i)
```

Procedury dźwiękowe

W **Action!** istnieją dwie procedury sterujące tworzeniem dźwięku: **Sound()** i **SndRst()**. Druga z nich jest bardzo łatwa do realizacji, natomiast znacznie gorzej jest z pierwszą. Wymaga ona bowiem czterech bajtów parametrów. Ponieważ zwykle generatorami dźwięku steruje się pojedynczo, to zamiast jednej procedury **Sound()** napiszmy cztery — osobno dla każdego generatora. Cały zestaw będzie następujący:

```
; procedury dzwiekowe

PROC SndRst()
 BYTE audctl=$D208,skctl=$D20F
  audctl=0
  skctl=3
RETURN

PROC Sound0(BYTE pit,dis,vol)
 BYTE audf=$D200,audc=$D201
  audf=pit
  audc=dis LSH 4 + vol
RETURN

PROC Sound1(BYTE pit,dis,vol)
 BYTE audf=$D202,audc=$D203
  audf=pit
  audc=dis LSH 4 + vol
RETURN

PROC Sound2(BYTE pit,dis,vol)
 BYTE audf=$D204,audc=$D205
  audf=pit
  audc=dis LSH 4 + vol
RETURN

PROC Sound3(BYTE pit,dis,vol)
 BYTE audf=$D206,audc=$D207
  audf=pit
  audc=dis LSH 4 + vol
RETURN
```

Jeżeli w pisanym programie konieczne będzie uzależnienie generatorów od siebie, to można to uzyskać zmieniając parametry przekazywane do procedury lub deklarując dodatkową zmienną globalną która określi numer generatora.

Procedury wejścia/wyjścia

Po tej prostej wprawce przejdźmy do jednej z trudniejszych (i ważniejszych) rzeczy: procedur obsługujących komunikację komputera z urządzeniami zewnętrznymi. Do ich definiowania wykorzystamy systemową procedurę **CIO**, która musi być następująco zadeklarowana na początku programu:

```
; glowna procedura we'jscia/wyjscia

PROC CIO=$E456(BYTE acc,xreg)
```

Najprostszą procedurą I/O *(Input/Output* — wejście/wyjście) jest **Close()**. Ponieważ dla zamknięcia kanału trzeba tylko wpisać odpowiedni rozkaz do bloku sterowania I/O, to narzuca się następujące rozwiązanie:

```
PROC Close(BYTE chn)
 BYTE iccmd=$342+(chn LSH 4)
  iccmd=$C
  CIO(0,chn LSH 4)
RETURN
```

Niestety, deklaracja zmiennej musi być wartością stałą. Zastosujmy więc rozwiązanie podobne jak w procedurach dźwiękowych. Nie trzeba oczywiście definiować ośmiu procedur, lecz tylko tyle, ile kanałów będziemy wykorzystywać w programie, na przykład:

```
PROC Close1()
 BYTE iccmd=$352
  iccmd=$C
  CIO(0,$10)
RETURN

PROC Close2()
 BYTE iccmd=$362
  iccmd=$C
  CIO(0,$20)
RETURN
```

Podobnie niezbędne będzie rozdzielenie na poszczególne kanały procedury **Open()**. Dodatkową trudność sprawia tu konieczność przekazania nazwy urządzenia. Najwygodniejsze jest więc przypisanie poszczególnych kanałów różnym urządzeniom, na przykład:

kanał	urządzenie
0	edytor
1	odczyt
2	zapis
4	drukarka
5	klawiatura
6	ekran

Teraz możemy definiować procedury w miarę potrzeby. Dla odczytu z klawiatury **Open()** może wyglądać następująco:

```
PROC OpenK()
 BYTE iccmd=$392,icax1=$39A
 CARD icba=$394,icbl=$398
 BYTE ARRAY name='K ': $9B
  iccmd=$3
  icba=name
  icbl=3
  icax1=4
  CIO(0,$50)
RETURN
```

a dla zapisu na drukarkę:

```
PROC OpenP()
 BYTE iccmd=$382,icax1=$38A
 CARD icba=$384,icbl=$388
 BYTE ARRAY name="P ": $9B
  iccmd=$3
  icba=name
  icbl=3
  icax1=8
  CIO(0,$40)
RETURN
```

W wypadku ekranu trzeba dodatkowo przekazać procedurze wartość trybu graficznego, w którym ma być utworzony obraz. Definicja procedury będzie więc nieco zmodyfikowana:

```
PROC OpenS(BYTE mode)
 BYTE iccmd=$3A2,icax1=$3AA,icax2=$3AB
 CARD icba=$3A4,icbl=$3A8
 BYTE ARRAY name="S ": $9B
  iccmd=$3
  icba=name
  icbl=3
  icax1=(mode&$F0)!$C
  icax2=mode
  CIO(0,$60)
RETURN
```

W zamieszczonej wyżej tabeli znajdują się pozycje "odczyt" i "zapis". Nie są to oczywiście urządzenia, lecz operacje wykonywane na pamięci zewnętrznej. Na przykład otwarcie magnetofonu do odczytu zrealizujemy następująco:

```
PROC OpenCR(BYTE gap)
 BYTE iccmd=$352,icax1=$35A,icax2=$35B
 CARD icba=$354,icbl=$358
 BYTE ARRAY name="C ": $9B
  iccmd=$3
  icba=name
  icbl=3
  icax1=4
  icax2=gap
  CIO(0,$10)
RETURN
```

Parametr "gap" oznacza w tej procedurze długość przerw między odczytywanymi rekordami. Wartość 0 oznacza długie przerwy, a 128 — krótkie. Analogicznie wykonamy odczyt do zapisu:

```
PROC OpenCW(BYTE gap)
 BYTE iccmd=$362,icax1=$36A,icax2=$36B
 CARD icba=$364,icbl=$368
 BYTE ARRAY name="C ": $9B
  iccmd=$3
  icba=name
  icbl=3
  icax1=8
  icax2=gap
  CIO(0,$20)
RETURN
```

Tą samą metodę można zastosować do plików dyskowych. Ponieważ nazwa pliku jest stosunkowo długa, to trzeba zastosować tu nieco inny sposób jej przekazania. Otwarcie do odczytu będzie więc następujące:

```
PROC OpenDR()
 BYTE iccmd=$352,icax1=$35A,icax2=$35B
 CARD icba=$354,icbl=$358
 BYTE ARRAY name
  name="D1:NAZWA.EXT"
  iccmd=$3
  icba=name(1)
  icbl=name(0)
  icax1=4
  CIO(0,$10)
RETURN
```

do zapisu zaś:

```
PROC OpenDW()
 BYTE iccmd=$362,icax1=$36A,icax2=$36B
 CARD icba=$364,icbl=$368
 BYTE ARRAY name
  name="D1:NAZWA.EXT"
  iccmd=$3
  icba=name(1)
  icbl=name(0)
  icax1=8
  CIO(0,$20)
RETURN
```

Dwie ostatnie z prezentowanych procedur mają zasadniczą wadę: dotyczą tylko jednego konkretnego pliku. Rozwiązanie takie jest do przyjęcia tylko w wypadku pliku o ustalonej nazwie, na przykład z danymi dla programu. Zwykle jednak występuje konieczność otworzenia pliku, którego nazwa jest nieznana w czasie pisania programu. Pożądane byłoby więc przekazywanie nazwy pliku do procedury, na przykład poprzez podanie adresu tablicy zawierającej tą nazwę. Dla odczytu może to wyglądać tak:

```
PROC OpenDR(CARD addr)
 BYTE iccmd=$352,icax1=$35A,icax2=$35B
 CARD icba=$354,icbl=$358
  iccmd=$3
  icba=addr+1
  icbl=addr
  icax1=4
  CIO(0,$10)
RETURN
```

a dla zapisu tak:

```
PROC OpenDW(CARD addr)
 BYTE iccmd=$362,icax1=$36A,icax2=$36B
 CARD icba=$364,icbl=$368
  iccmd=$3
  icba=addr+1
  icbl=addr
  icax1=8
  CIO(0,$20)
RETURN
```

Zastosowanie tego sposobu wymaga zadeklarowania na początku programu globalnej tablicy znakowej, w której będzie przechowywana nazwa pliku (oczywiście tablic takich może być kilka). Na przykład:

```
BYTE ARRAY name
```

Przed otwarciem kanału odpowiednią nazwę pliku należy umieścić w tablicy, a następnie wywołać właściwą procedurę:

```
name="D1:NAZWA1.EXT"
OpenDR(name)
name="D1:NAZWA2.EXT"
OpenDW(name)
```

Procedury zapisu

Korzystając z opisanego wyżej schematu, można tworzyć kolejne procedury. Jako pierwszą zrealizujmy procedurę **Put()**, która zapisuje jeden bajt do edytora (kanał 0), w odróżnieniu od normalnej wersji zapisującej do urządzenia określonego przez zmienną "device".

```
PROC Put(BYTE chr)
 BYTE iccmd=$342
 CARD icbl=$348
  iccmd=$B
  icbl=0
  CIO(chr,$00)
RETURN
```

Możemy teraz zastosować ją w następnej procedurze, która umieszcza w edytorze znak końca wiersza:

```
PROC PutE()
  Put($9B)
RETURN
```

UWAGA: Kanał 0 jest automatycznie otwierany dla edytora przez system operacyjny i nie powinien być zamykany. Procedury **Open()** i **Close()** dla tego kanału są więc zbędne.

W identyczny sposób definiujemy procedury **Put()** dla pozostałych urządzeń wykorzystywanych przez program. Na przykład dla drukarki (kanał 4) będą one następujące:

```
PROC PutP(BYTE chr)
 BYTE iccmd=$382
 CARD icbl=$388
  iccmd=$B
  icbl=0
  CIO(chr,$40)
RETURN

PROC PutPE()
  PutP($9B)
RETURN
```

W celu realizacji procedury **Print()** trzeba przekazać do bloku sterowania I/O adres napisu i jego długość. Jednakże prostsze (choć nieco dłuższe) będzie zastosowanie tu procedury **Put()** umieszczonej w pętli. Dla określenia wartości granicznej pętli użyjemy przy tym pierwszego elementu tablicy znakowej, który określa jej długość.

```
PROC Print(BYTE ARRAY txt)
 BYTE i
  FOR i=1 TO txt(0)
   DO
    Put(txt(i))
   OD
RETURN
```

Przez proste dodanie na końcu wywołania **PutE()** otrzymamy procedurę **PrintE()**. Aby jednak nie pisać jej od nowa, skorzystamy z uprzednio zdefiniowanej procedury **Print()**:

```
PROC PrintE(BYTE ARRAY txt)
  Print(txt)
  PutE()
RETURN
```

Powyższych procedur nie można jednak wykorzystać do zapisu wartości liczbowych. Podanie liczby typu BYTE spowoduje bowiem zapisanie znaku o odpowiadającym jej kodzie, a dla liczb CARD i INT będą to dwa znaki. Konieczne jest zatem uprzednie przekształcenie liczby na znaki kodu ASCII (cyfry), które będą umieszczone w tablicy tekstowej. Ponieważ tablica nie może być wartością zwracaną przez funkcję, to zamiana musi być dokonywana przez procedurę. Adres tablicy może być jednak parametrem procedury tylko w wypadku zamiany liczby typu BYTE. Równoczesne przekazanie dwubajtowej liczby CARD lub INT i wskaźnika tablicy (również dwa bajty) jest niemożliwe, gdyż przekracza dopuszczalną liczbę parametrów. Jedynym wyjściem jest więc zadeklarowanie globalnej tablicy, w której będą umieszczane przekształcone liczby:

```
BYTE ARRAY numstr
```

Aby utrzymać jednolitą strukturę procedur i uniknąć pomyłek, zastosujemy identyczny schemat dla wszystkich typów zmiennych, pomimo iż dla typu BYTE nie jest to konieczne. Odpowiednia procedura przekształcania liczby BYTE będzie więc miała następującą postać:

```
PROC StrB(BYTE val)
 BYTE tmp
  IF val=0 THEN
    numstr="0"
    RETURN
  FI
  tmp=val
  numstr="000"
  WHILE tmp>99
   DO
    numstr(1)==+1
    tmp==-100
   OD
  WHILE tmp>9
   DO
    numstr(2)==+1
    tmp==-10
   OD
  WHILE tmp>0
   DO
    numstr(3)==+1
    tmp==-1
   OD
  WHILE numstr(1)='0
   DO
    numstr(1)=numstr(2)
    numstr(2)=numstr(3)
    numstr(0)==-1
   OD
RETURN
```

Mając już liczbę zapisaną w postaci ciągu znaków, możemy ją wyświetlić za pomocą procedur:

```
PROC PrintB(BYTE num)
  StrB(num)
  Print(numstr)
RETURN

PROC PrintBE(BYTE num)
  PrintB(num)
  PutE()
RETURN
```

Przekształcenie liczby typu CARD jest realizowane w ten sam sposób jak BYTE, lecz — ze względu na większą długość liczby — w większej liczbie kroków. Wynik przekształcenia jest również umieszczany w tablicy "numstr":

```
PROC StrC(CARD val)
 CARD tmp
  IF val=0 THEN
    numstr="0"
    RETURN
  FI
  tmp=val
  numstr="00000"
  WHILE tmp>9999
   DO
    numstr(1)==+1
    tmp==-10000
   OD
  WHILE tmp>999
   DO
    numstr(2)==+1
    tmp==-1000
   OD
  WHILE tmp>99
   DO
    numstr(3)==+1
    tmp==-100
   OD
  WHILE tmp>9
   DO
    numstr(4)==+1
    tmp==-10
   OD
  WHILE tmp>0
   DO
    numstr(5)==+1
    tmp==-1
   OD
  WHILE numstr(1)='0
   DO
    numstr(1)=numstr(2)
    numstr(2)=numstr(3)
    numstr(3)=numstr(4)
    numstr(4)=numstr(5)
    numstr(0)==-1
   OD
RETURN
```

Uzyskaną w rezultacie przekształcenia liczbę możemy już wyświetlić za pomocą procedur:

PROGRAMOWANIE

```
PROC PrintC(CARD num)
  StrC(num)
  Print(numstr)
RETURN

PROC PrintCE(CARD num)
  PrintC(num)
  PutE()
RETURN
```

Liczba typu INT różni się od CARD tylko sposobem interpretacji zapisanej wartości. Przy przekształcaniu jej na ciąg znaków można więc wykorzystać zdefiniowaną już procedurę **StrC()**. Dla dodatnich wartości liczby INT przekształcenie będzie identyczne jak CARD, dla wartości ujemnych zaś przed przekształceniem należy zmienić znak liczby, a potem dopisać do uzyskanego ciągu znak "-".

```
PROC StrI(INT val)
 CARD tmp
 BYTE i
  IF val=0 THEN
    numstr="0"
    RETURN
  FI
  IF val>0 THEN
    tmp=val
    StrC(tmp)
    RETURN
  FI
  tmp=-val
  StrC(tmp)
  numstr(0)==+1
  FOR i=numstr(0) TO 2
   DO
    numstr(i)=numstr(i-1)
   OD
  numstr(1)='-
RETURN
```

Procedury **PrintI()** i **PrintIE()** będą oczywiście analogiczne do pokazanych wyżej:

```
PROC PrintI(INT num)
  StrC(num)
  Print(numstr)
RETURN

PROC PrintIE(INT num)
  PrintI(num)
  PutE()
RETURN
```

Wszystkie pokazane wyżej procedury powodują zapisanie wskazanych wartości w edytorze, a więc na ekranie, dla innych urządzeń (drukarka, magnetofon i stacja dyskow) należy zdefiniować podobne procedury. Oczywiście w pisanym programie wystarczy umieścić tylko te, które są niezbędne. Na przykład, nie trzeba pisać procedur dla drukarki, jeśli nie przewiduje się w programie wykonywania wydruków.

Procedury odczytu

Procedury służące do odczytu danych skonstruujemy w taki sam sposób jak procedury zapisu. Także i w tym wypadku zostaną zaprezentowane tylko procedury odczytu z jednego urządzenia, zdefiniowanie zaś pozostałych pozostanie dla czytelników. Tym wybranym urządzeniem jest klawiatura, która w odróżnieniu od edytora musi być uprzednio otwarta w programie (procedurą **OpenK()**.

Podstawową procedurą odczytu jest **Get()**. W jej definicji konieczne jest użycie krótkiego bloku kodu maszynowego. Ponieważ procedura **CIO** nie zwraca żadnej wartości, a po wykonaniu przez nią operacji GET odczytany bajt znajduje się w akumulatorze, to zapisujemy bezpośrednio po **CIO** rozkaz STA i adres zmiennej "val" (typu BYTE). Cała funkcja **Get()** wygląda następująco:

```
CHAR FUNC Get()
 BYTE val,iccmd=$392
 CARD icbl=$398
  iccmd=$7
  icbl=0
  CIO(0,$50)
  [$8D val]
RETURN(val)
```

Przez wywoływanie funkcji **Get()** w pętli, aż do odczytania znaku końca wiersza ($9B), utworzymy procedurę **InputS()** odczytującą ciąg znaków:

```
PROC InputS(BYTE ARRAY txt)
 BYTE i=[0],k
  k=Get()
  WHILE k#$9B
   DO
    i==+1
    txt(i)=k
    k=Get()
   OD
  txt(i)=k
  txt(0)=i
RETURN
```

Przy odczycie liczb stajemy przed problemem odwrotnym niż przy zapisie. Ponieważ jednak liczba może być zwracana przez funkcję, to w takiej właśnie formie zdefiniujemy przekształcenia ciągów na liczby. Pierwszą taką funkcją będzie oczywiście **ValB()**:

```
BYTE FUNC ValB(BYTE ARRAY str)
 BYTE i,j,num=[0]
  FOR i=1 TO str(0)
   DO
    j=num
    j==LSH 2 + num
    num=j LSH 1
    num==+str(i)-$30
   OD
RETURN(num)
```

Korzystając z tego przekształcenia oraz z procedury **InputS()** możemy teraz zbudować funkcję odczytującą wartości typu BYTE:

```
BYTE FUNC InputB()
 BYTE val
 BYTE ARRAY txt
  InputS(txt)
  val=ValB(txt)
RETURN(val)
```

Niemal identyczna z **ValB()** jest funkcja **ValC()**. Jednakże musi ona być zdefiniowana oddzielnie, ze względu na inny typ występującej w niej wartości.

```
CARD FUNC ValC(BYTE ARRAY str)
 BYTE i
 CARD j,num=[0]
  FOR i=1 TO str(0)
   DO
    j=num
    j==LSH 2 + num
    num=j LSH 1
    num==+str(i)-$30
   OD
RETURN(num)
```

W ten sam sposób **InputC()** różni się od **InputB()**:

```
CARD FUNC InputC()
 CARD val
 BYTE ARRAY txt
  InputS(txt)
  val=ValC(txt)
RETURN(val)
```

Przy realizacji funkcji **ValI()** wykorzystamy (podobnie jak przy **StrI()**) funkcję **ValC()**. Po rozpoznaniu znaku minus na początku ciągu zostaje on usunięty i liczba jest przekształcana jak liczba CARD, a w wyniku zmieniany jest znak. Jeśli liczba jest dodatnia, to jest przekształcana jak CARD.

```
INT FUNC ValI(BYTE ARRAY str)
 BYTE i
 INT num
  IF str(1)='- THEN
    FOR i=2 TO str(0)
     DO
      str(i-1)=str(i)
     OD
    str(0)==-1
    num=-ValC(str)
  ELSE
    num=ValC(str)
  FI
RETURN(num)
```

Tak zdefiniowaną funkcję **ValI()** użyjemy do określenia funkcji **InputI()**:

```
INT FUNC InputI()
 INT val
 BYTE ARRAY txt
  InputS(txt)
  val=ValI(txt)
RETURN(val)
```

UWAGA: W odróżnieniu od normalnych procedur **Input()** w **Action!** procedury zdefiniowane powyżej korzystają z kanału 5. Musi on więc zostać otwarty przed użyciem którejkolwiek z nich.

Procedury graficzne

W zasadzie procedury graficzne są także procedurami wejścia/wyjścia. Zostały one jednak wyodrębnione, ponieważ korzystają zawsze z kanału 6. Ponadto znakiem zapisywanym przez te procedury jest kod koloru określony przez zadeklarowaną na początku programu zmienną.

```
BYTE color=$2FD
```

W **Action!** zmienna ta ma zawsze początkową wartość równą 0. Nie można jednak w deklaracji zmiennej określić równocześnie jej adresu i wartości, więc należy to wykonać w początkowej części programu lub bezpośrednio przed użyciem jednej z procedur rysujących na ekranie.

Zanim zostanie wykonana jakakolwiek procedura rysująca na ekranie lub odczytująca z niego dane, musi być otwarty przeznaczony do tego kanał. Realizuje to procedura **Graphics()**, której parametrem jest tryb graficzny, w jakim będzie otwarty ekran. Procedura ta jest właściwie sekwencją procedur **Close()** i **Open()**. Ponieważ jednak nie stosuje się oddzielnie procedury **Close()** dla kanału 6, to została ona tu włączona. W rezultacie procedura **Graphics()** wygląda następująco:

```
PROC Graphics(BYTE mode)
 BYTE iccmd=$3A2
  iccmd=$C
  CIO(0,$60)
  OpenS(mode)
RETURN
```

Jeżeli nie przewidujemy w programie samodzielnego stosowania procedury **OpenS()** (czyli niemal zawsze), to można jej definicję zapisać bezpośrednio w definicji **Graphics()**.

Narysowanie punktu jest równoważne zapisaniu pojedynczego bajtu w określonym miejscu ekranu. Można więc zrealizować to przez połączenie procedur **Position()** i **Put()**. Wymagałoby to jednak zdefiniowania procedury **Put()** dla kanału 6. Zamiast tego odpowiednie instrukcje zapiszemy od razu w procedurze **Plot()** (właśnie tak można wykonać również **Graphics()**:

```
PROC Plot(CARD col BYTE row)
 BYTE iccmd=$3A2
 CARD icbl=$3A8
  Position(col,row)
  iccmd=$B
  icbl=0
  CIO(color,$60)
RETURN
```

Rysowanie linii jest natomiast realizowane przez specjalną procedurę systemu operacyjnego. Jest ona wywoływana przez procedurę **CIO**, jeśli kod rozkazu wynosi $11. Przedtem trzeba jeszcze umieścić kursor na końcu linii i zapisać kolor w rejestrze ATACHR. W efekcie otrzymujemy następującą procedurę:

```
PROC DrawTo(CARD col BYTE row)
 BYTE atachr=$2FB,
      iccmd=$3A2,icax1=$3AA
  Position(col,row)
  atachr=color
  iccmd=$11
  icax1=$C
  CIO(0,$60)
RETURN
```

Ta sama procedura systemu realizuje wypełnianie obszaru, jeśli kod rozkazu będzie wynosił $12 (dziwne, że nie wykorzystali tego autorzy Atari Basic). Po zmianie tej wartości uzyskamy procedurę **Fill()**:

```
PROC Fill(CARD col BYTE row)
 BYTE atachr=$2FB,
      iccmd=$3A2,icax1=$3AA
  Position(col,row)
  atachr=color
  iccmd=$12
  icax1=$C
  CIO(0,$60)
RETURN
```

Funkcja **Locate()** służy do odczytu informacji z ekranu. Różni się ona od **Get()** dokładnie tak samo, jak **Plot()** różni się od **Put()**. Bez dalszych komentarzy można więc przedstawić jej definicję:

```
BYTE FUNC Locate(CARD col BYTE row)
 BYTE val,iccmd=$3A2
 CARD icbl=$3A8
  Position(col,row)
  iccmd=$7
  icbl=0
  CIO(0,$60)
  [$8D val]
RETURN(val)
```

Inne procedury

Pozostały jeszcze nie zdefiniowane procedury operujące na blokach pamięci (**MoveBlock()**, **SetBlock()** i **Zero()**) oraz tablicach znakowych (**SAssign()**, **SCopy()**, **SCopyS()** i **SCompare()**), także procedury **Poke()** i **Peek()**. Ostatnie z wymienionych są praktycznie zbędne i dlatego zostały pominięte.

Inaczej przedstawia się sprawa procedur operujących na blokach i tablicach. Wymagają one tak wielu parametrów, że trzeba je definiować indywidualnie do konkretnych zastosowań albo przekazywać niezbędne parametry przy użyciu zmiennych globalnych. Ponieważ poza tym są one bardzo proste, to uznałem, że każdy programista napisze je w takiej wersji, jaka będzie najbardziej odpowiednia dla tworzonego przezeń programu.

Wojciech ZIENTARA

Efektem kompilacji programu napisanego w Action! jest program w języku maszynowym. Jeśli podczas jego pisania przestrzegaliśmy zasad podanych w poprzednim odcinku, to program ten będzie działał także bez samego Action!. Taki program maszynowy można dołączyć do innego programu stworzonego w każdym niemal języku programowania. Na przykład, w Turbo Basicu.

Podczas kompilacji **Action!** sam określa adres początkowy programu w pamięci. Istnieje jednak sposób ustalenia tego adresu przez użytkownika. Należy tylko wpisać go do komórek pamięci o adresach $0E i $0491. Najlepiej wykonać to za pomocą dyrektywy SET, użytej nie jako rozkaz monitora, lecz jako dyrektywa kompilatora. Na przykład:

```
SET $E=20000
SET $491=20000

BYTE blablabla
CARD blablablabla

PROC JAKAS_TAM()
```

W ten sposób skompilowany program rozpocznie się od adresu 20000. Pamiętaj, że oba wskazane adresy muszą mieć tą samą wartość. Rejestr $0E zawiera adres kompilacji, a $0491 oznacza początek pamięci zapisywanej poleceniem monitora „W".

Są jeszcze trzy rzeczy, o których należy koniecznie pamiętać łącząc **Action!** z innym językiem programowania:

OBSZAR PAMIĘCI

Skompilowany program w **Action!** musi zajmować obszar pamięci niewykorzystywany przez język, z którym go łączymy. Dolna granica pamięci RAM dostępnej dla użytkownika jest przechowywana w rejestrze MEMLO (komórki 743 i 744), górna zaś w MEMTOP (741 i 742). Zawartość MEMTOP zmienia się i zależy od wybranego trybu graficznego. W obszarze tym musi znaleźć się cały program. Dla programu w **Action!** pozostaje więc obszar między końcem programu w innym języku, a MEMTOP. Na przykład, adres końca programu w **Basicu** lub **Turbo Basicu** jest przechowywany w komórkach 144 i 145 (BMEMH). Od tego adresu do MEMTOP znajduje się obszar, który może być przeznaczony dla **Action!**. BMEMH jednak również ulega zmianie, gdy deklarujemy nowe tablice, wywołujemy procedury lub otwieramy pętle.

Z tych powodów najlepszym rozwiązaniem jest zarezerwowanie dla **Action!** specjalnego obszaru pamięci. Realizuje się to podobnie jak przy rezerwowaniu miejsca dla grafiki graczy i pocisków lub zestawów znaków. Korzysta się w tym celu z rejestru RAMTOP (106), w którym znajduje się numer najwyższej strony pamięci stojącej do dyspozycji. Wystarczy tylko zmienić jego zawartość i wykonać instrukcję GRAPHICS (w celu przesunięcia pamięci obrazu). W **Turbo Basicu** na przykład:

POKE 106, 160: GRAPHICS 0

Teraz mamy do dyspozycji bezpieczny obszar od 40960 do 49152 (160*256 = 40960). Do tego obszaru można już bezpośrednio wczytać program w **Action!**.

Pojawia się tu wszakże kolejny problem: programu w **Action!** nie można skompilować w takim obszarze pamięci, gdyż jest on zajmowany przez moduł **Action!** — jest to możliwe tylko w wersji dyskowej lub kasetowej. Można to jednak rozwiązać korzystając z tzw. kompilacji OFFCODE. Na czym to polega? Program można skompilować w normalnie dostępnym obszarze pamięci, lecz w taki sposób, że będzie on później umieszczany w innym obszarze. Do określenia przesunięcia programu wynikowego służą komórki $B5 i $B6. Musi być w nich wpisana różnica między właściwym adresem umieszczenia programu, a adresem, od którego będzie on kompilowany. Na przykład, program powinien być umieszczony od adresu $A000, ale podczas kompilacji będzie zapisywany od adresu $5000. Trzeba więc na początku programu napisać:

```
SET $E=$5000
SET $491=$5000
SET $B5=$5000
```

gdyż $5000 + $5000 = $A000. Gdy rejestr $B5 nie był ustawiany, to znajduje się w nim zawsze wartość 0.

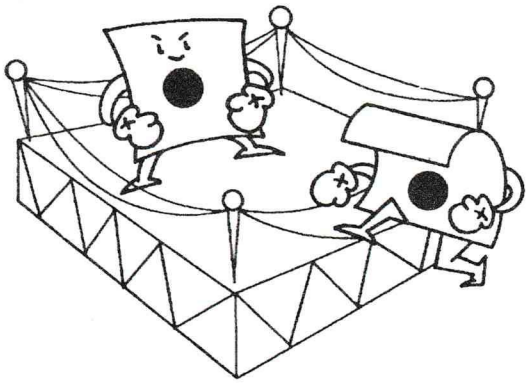
DZIAŁANIE PROGRAMU

Jeżeli nie posiadamy biblioteki OSS Runtime Library, to program w **Action!** będzie działał tylko wtedy, gdy zostanie napisany zgodnie z zasadami opisanymi w poprzednim „Moim Atari" („Action! bez Action! "). Muszą być więc spełnione następujące warunki:

a) program nie wykorzystuje żadnych PROCedur bibliotecznych **Action!**;
b) do każdej procedury nie jest przekazywane więcej niż trzy bajty parametrów (na wszelki wypadek lepiej przekazywać tylko dwa bajty);
c) w programie nie ma żadnego mnożenia i dzielenia;
d) operatory przesunięcia (RSH i LSH) nie są używane z wartościami typu CARD i INT.

URUCHOMIENIE ACTION!

W większości wypadków zapisuje się skompilowany program w **Action!** rozkazem „W" monitora. Na końcu pliku jest wtedy zapisywany jako ostatni segment adres uruchomienia (INIT). Adres ten (komórki 738 i 739) wskazuje, ostatnią PROCedurę w programie. Zapisany w ten sposób program będzie automatycznie uruchomiony po każdym wczytaniu do komputera. W **Turbo Basicu** do wczytania takiego programu trzeba użyć instrukcji BRUN lub BLOAD



— i zawsze nastąpi samoczynne uruchomienie wczytanego programu.

Niestety, nie ma takiej możliwości w **Atari Basic**. Wczytany program trzeba więc uruchamiać instrukcją USR. Pojawia się tu jednak dodatkowy kłopot, gdyż USR zapisuje na stosie liczbę przekazywanych parametrów. Nawet, gdy ich nie ma, zapisywana jest wartość 0. Normalnie więc wszystkie procedury wywoływane przez USR zaczynają się od rozkazu PLA, który zdejmuje ze stosu bajt określający liczbę parametrów. Jest to konieczne dla zapewnienia poprawnej pracy programu. Co w takim przypadku zrobić w **Action!**? Nie ma tam przecież instrukcji PLA.

To żaden kłopot! W **Action!** można dodawać fragmenty programu maszynowego jako bloki kodu. Są to ograniczone nawiasami kwadratowymi liczby odpowiadające rozkazom procesora. Trzeba więc tylko dodać jako blok kodu wartość 104, która odpowiada rozkazowi PLA. Na przykład:

```
PROC XY()

[104]

RETURN
```

Można też oczywiście wpisać jako blok kodu większy fragment programu maszynowego: [104 169 1 ...]. Ponadto w bloku kodu można wpisać zmienne **Action!** zamiast bezpośrednich adresów.

[104] najlepiej jest umieścić na samym początku ostatniej PROCedury. Zapewnia to bezbłędne działanie programu i umożliwia w razie konieczności powrót do **Basica**.

Niezbędna jest jeszcze znajomość adresu początkowego ostatniej PROCedury, aby podać go w instrukcji USR. Adres początkowy, który był użyty w dyrektywie SET i został wpisany do rejestrów $0E i $0491, NIE jest adresem ostatniej procedury. Trzeba więc skompilować program w **Action!**, a następnie odczytać adres ostatniej PROCedury za pomocą rozkazu „?" monitora wpisując:

?nazwa_PROC

Opisane tu możliwości połączenia **Action!** z innymi językami programowania z pewnością umożliwią zwiększenie atrakcyjności programów i ułatwią ich tworzenie.

Wojciech ZIENTARA

Nabywcy dyskietki z programami opublikowanymi w tym numerze „Mojego Atari" otrzymają jako premię dwa programy ilustrujące opisywane tu łączenie **Turbo Basica** i **Action!**. Programy te pochodzą z niemieckiego pisma „Atari Magazin".

PROGRAMOWANIE

Action! + BASIC

RUCH STATKU

Do tej pory projektowaliśmy głównie operacje stanowiące przygotowanie do właściwej rozgrywki. Dopiero teraz możemy zająć się zasadniczą częścią naszego programu.

Przypomninam, że gra właściwa jest pętlą, w której kolejno są wywoływane następujące procedury:

Ruch Statku
Ruch Niszczyciela
Ruch Okrętu Podwodnego
Odpalenie Torpedy
Peryskop
Ruch Okrętu Podwodnego
Peryskop

Procedury te są znacznie bardziej skomplikowane od dotychczas omawianych, więc będziemy musieli każdą z nich przed zakodowaniem rozwinąć w schemat blokowy. Taki tok postępowania ułatwi zapisanie programu w formie zrozumiałej dla komputera.

Pierwszym elementem tego cyklu jest procedura realizująca przemieszczanie się statku. Należy więc obliczyć kierunek ruchu tak, aby statek nie opuścił planszy i odpowiednio zmodyfikować zmienne opisujące jego położenie. Konieczne jest przy tym sprawdzenie, czy statek nie znajdzie się na polu zajętym już przez jeden z niszczycieli. W takim przypadku faza obliczania parametrów ruchu statku musi być powtórzona.

Po wykonaniu przez statek ruchu może on znaleźć się nad okrętem podwodnym. Sytuacja taka z pewnością będzie rozpoznana przez okręt podwodny, więc odpowiednia informacja musi zostać wyświetlona na ekranie. Drugi — ważniejszy — przypadek to znalezienie się statku w porcie. Powoduje to automatycznie zakończenie gry. Sygnalizujemy to nadając ustaloną wartość (tu 4) zmiennej „q" i opuszczając procedurę.

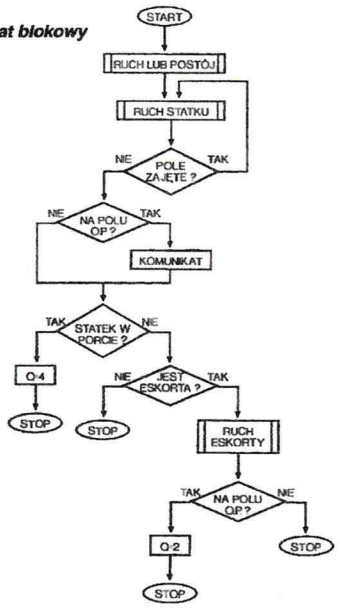
Następnie należy sprawdzić, czy nie został uprzednio zatopiony niszczyciel eskortujący. Jeżeli jeszcze istnieje, to ustalamy jego nowe położenie w pobliżu statku (na polu sąsiadującym). Teraz również trzeba sprawdzić, czy na tym samym polu nie znajduje się okręt podwodny. Jeżeli tak, to zakładamy, że niszczyciel wykrywa szykujący się do ataku okręt i niszczy go bombami głębinowymi. Sygnalizujemy tą ewentualność nadając zmiennej „q" wartość 2 i kończąc procedurę.

Ponieważ statek dążąc do portu mógłby zbyt szybko umknąć nie dając szans na zniszczenie, należy wprowadzić dodatkowe ograniczenie. Będzie ono polegać na tym, że statek przybywając na redę portu może oczekiwać na miejsce i w związku z tym przez kilka (1—3) faz nie wykonuje ruchu. Po uwzględnieniu tego ograniczenia schemat blokowy naszej procedury będzie wyglądał jak na rys. 1.

Rozwijając ten schemat należy przede wszystkim pamiętać o wprowadzeniu warunków sprawdzających, czy któraś z jednostek nie znalazła się poza planszą gry. Jeśli taki przypadek zachodzi można zmienić kierunek ruchu na przeciwny (tak jest dla statku) lub powtórzyć całe obliczanie nowej pozycji (tak jest dla niszczyciela eskorty). Różne podejście do tego zagadnienia wynika z faktu, że statek porusza się po planszy swobodnie, niszczyciel zaś jest związany ze statkiem i zawsze musi się znajdować w jego pobliżu.

Uważny czytelnik analizując wydruki procedur w **Action!** (listing 1) lub w **Basicu** (listing 2) zauważy z pewnością, że ruch statku jest poddany dodatkowemu ograniczeniu. Aby statek nie dążył do portu najkrótszą możliwą drogą, zostało bowiem dodane losowe (z prawdopodobieństwem 25%) zmienianie kierunku ruchu w pionie i w poziomie.

Rys. 1
Schemat blokowy



Szybkość pracy komputera znacznie przekracza możliwości percepcji człowieka. Aby możliwe było odczytanie wyświetlanych przez program komunikatów, należy wprowadzić procedurę wstrzymującą na pewien okres czasu wykonywanie programu. Procedura ta jest w **Action!** następująca:

```
PROC Opoznienie(CARD i)
 CARD j
  FOR j=1 TO i DO OD
RETURN
```

Została ona zrealizowana w najprostszy możliwy sposób, czyli przez zastosowanie pętli. Dla umożliwienia regulacji opóźnienia granicę pętli podaje się jako parametr procedury. Nieco inaczej wygląda to w **Basicu**, gdyż nie można tam przekazać parametru:

```
RK    108 REM
QH    109 REM *** Opoznienie ***
GS    110 FOR L=1 TO DL:NEXT L:RETURN
```

W tym przypadku długość opóźnienia musi być ustalona przed wywołaniem procedury przez nadanie odpowiedniej wartości zmiennej „DL".

Porównując wywołania procedury opóźniającej w obu stosowanych językach łatwo można zauważyć znaczną różnicę wartości parametru. Wynika ona z różnej szybkości działania **Basica** i **Action!.** Nie oznacza to oczywiście, że program w **Action!** jest ponad 100 razy szybszy. Jednak w przypadku pętli taka różnica występuje. Pojawia się jeszcze pytanie, jak określić wartości opóźnienia. Można próbować je wyliczyć, ale najprostszym sposobem jest doświadczalne ustalenie podczas próbnego uruchamiania programu.

Wojciech ZIENTARA

LISTING 1

```
PROC RuchStatku()
 BYTE s,n,k,nx,ny
  IF sx<2 AND sy<2 THEN
    s=-1
  ELSE
   n=Rand(100)
   IF n>40 THEN
     s=3
   ELSEIF n>20 THEN
     s=2
   ELSEIF n>5 THEN
     s=1
   FI
   Czyszczenie()
   PrintE("Statek wykonal ruch.")
   Opoznienie(30000)
  FI
  IF s=0 THEN
    RETURN
  FI
  DO
  nx=sx ny=sy
    IF Rand(100)<75 AND nx>0 OR nx>8 THEN
      nx=-1
    ELSE
      nx=+1
    FI
    IF Rand(100)<75 AND ny>0 OR ny>8 THEN
      ny=-1
    ELSE
      ny=+1
    FI
    IF (nx=dx(1) OR ny=dy(1)) AND
       (nx=dx(2) OR ny=dy(2)) THEN
      EXIT
    FI
  OD
  ox=sx oy=sy sx=nx sy=ny
  Kasowanie(ox,oy)
  IF sx=mx AND sy=my THEN
    PrintE("Statek jest nad okretem podwodnym."
    Opoznienie(30000)
  FI
  IF sx=0 AND sy=0 THEN
    q=4 RETURN
  FI
  IF ns(2)#0 THEN
    Kasowanie(dx(2),dy(2))
    DO
      dx(2)=sx+1-2*Rand(2)
    UNTIL dx(2)<10 OD
    DO
      dy(2)=sy+1-2*Rand(2)
    UNTIL dy(2)<10 OD
    IF dx(2)=mx AND dy(2)=my THEN
      Opoznienie(10000)
      q=2
    FI
  FI
RETURN
```

LISTING 2

```
SC    497 REM
JL    498 REM *** Ruch Statku ***
SI    499 REM
WW    500 IF SX<2 AND SY<2 THEN S=S-1:GOTO 5
      20
OT    510 N=RND(0):S=(N>0.4)+(N>0.2)+(N>0.05
      )
KM    520 GOSUB 40:? "Statek wykonal ruch.":
      DL=200:GOSUB 110
UX    530 IF S=0 THEN RETURN
XO    540 NX=SX:NY=SY:CX=1:CY=1
HD    550 IF RND(0)<0.75 AND NX>0 OR NX>8 TH
      EN CX=-1
QY    560 NX=NX+CX:IF RND(0)<0.75 AND NY>0 O
      R NY>8 THEN CY=-1
HI    570 NY=NY+CY:IF NX=DX(0) AND NY=DY(0)
      OR NX=DX(1) AND NY=DY(1) THEN 540
OE    580 OX=SX:OY=SY:SX=NX:SY=NY:GOSUB 50
LK    590 IF SX=MX AND SY=MY THEN ? "Statek
      jest nad okretem podwodnym.":DL=200:GO
      SUB 110
JH    600 IF SX=0 AND SY=0 THEN Q=4:RETURN
QC    610 IF NS(1)=0 THEN RETURN
QK    620 OX=DX(1):OY=DY(1):GOSUB 50
RM    630 DX(1)=SX+1-2*INT(RND(0)*2):ON DX(1
      )<0 OR DX(1)>9 GOTO 630
XT    640 DY(1)=SY+1-2*INT(RND(0)*2):ON DY(1
      )<0 OR DY(1)>9 GOTO 640
BN    650 IF DX(1)=MX AND DY(1)=MY THEN DL=1
      00:GOSUB 110:Q=2
ZO    660 RETURN
```