PROGRAMOWANIE MASZYN CYFROWYCH
FORTRAN 90/95

NOTATKI DO WYKŁADU
z wykorzystaniem kompilatora Essential Lahey Fortran 90 ELF90 (wersja freeware)

Podstawowy cel wykładu:

- zapoznanie z językiem programowania wysokiego poziomu Fortran 90/95;
- zapoznawanie z podstawowymi metodami projektowania, pisania oraz uruchamiania własnych programów w języku programowania Fortran 90/95;

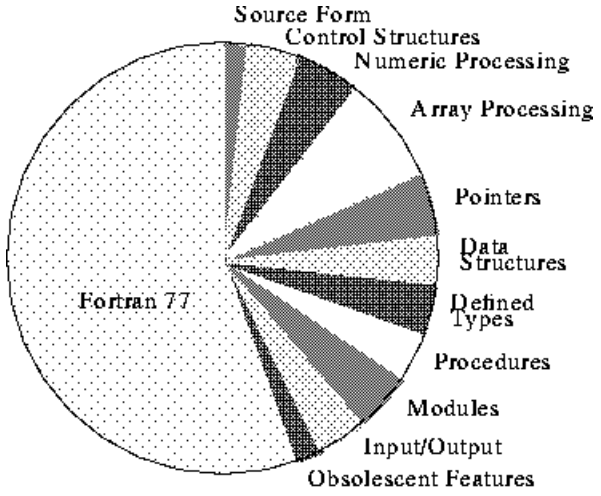
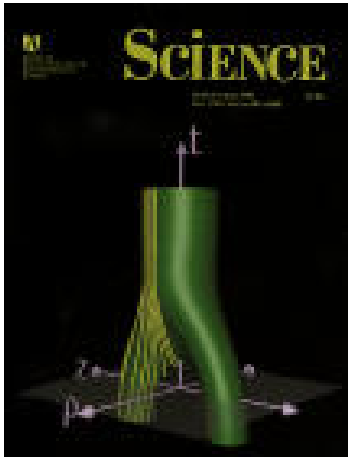


Figure 1 Fortran 90.

DEFINICJE I POJĘCIA PODSTAWOWE



Informacja - obiekt abstrakcyjny, który w postaci zakodowanej (jako tzw. **dana**) może być przechowywana, przesyłana, przetwarzana lub też użyta do sterowania.

Dane - informacje wyrażone w pewnym języku naturalnym lub sztucznym. Dane mogą być **proste** lub **złożone.** Dane złożone składają się z danych prostych o określonym wzajemnym usytuowaniu i z określonymi regułami dostępu. W informatyce dane są obiektami, na których operują programy.

Informatyka - ogół dyscyplin naukowych i technicznych zajmujących się *przetwarzaniem informacji za pomocą komputerów*.

Teorie informatyczne zajmują się badaniem zjawisk związanych z operowaniem informacją: jej przedstawianiem, przechowywaniem i przetwarzaniem.

Informatyka zajmuje się zarówno sprzętem komputerowym jak i narzędziami takimi jak algorytmy, języki programowania itd.

MASZYNY CYFROWE (KOMPUTERY) SĄ TO MASZYNY PRZEZNACZONE DO AUTOMATYCZNEGO PRZETWARZANIA INFORMACJI WEDŁUG ZADANEGO ALGORYTMU

PROGRAM KOMPUTEROWY JEST ZAKODOWANĄ INFORMACJĄ

(Pre)Historia maszyn cyfrowych

Starożytność - *abakus* (liczydło)

calculi - kamyczki służące Rzymianom do obliczeń w abakusie.

compute (późna łacina) - nacinanie nacięć/karbów na drzewie.

Cyfry - znaki służące do zapisywania liczb;

Starożytność	- znaki proste (kreski) dla poszczególnych jednostek;
Cyfry rzymskie	- pochodzenia etruskiego (ok. 500 p.n.e.);
X-III w.	- przeniesienie cyfr arabskich do Europy (pochodzą z Indii);
XV w.	- rozpowszechnienie cyfr arabskich w Europie;

1631 - **W. Schickard** (prof. języków biblijnych i astronomii w Tybindze, 1592-1635) - pierwsza maszyna do dodawania z przeniesieniami.

1642(5) - **B. Pascal** (1623-1662) - sumator szeregowy (tylko dod. i odejm.)

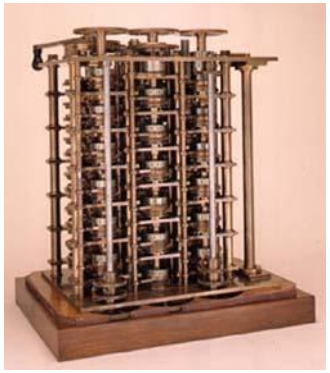
(1666 Morland, 1678 Grillet, 1722 Gerten)

1671 - **G. Leibnitz** (1646-1716) - sumator równoległy (tylko dod. i odejm.)

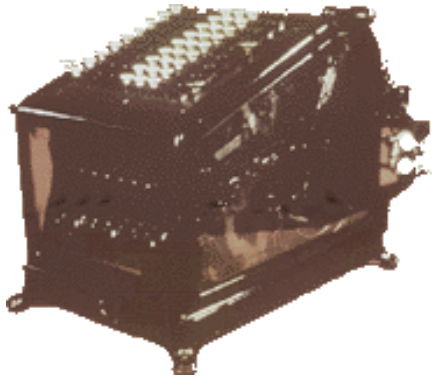
1679 - Leibnitz opisuje obliczenia w dwójkowym systemie zapisu liczb (Bacon) i proponuje budowę maszyny liczącej w tym systemie
1694 - Leibnitz konstruuje arytmometr 4-działaniowy z walcami schodkowymi.

1741 - Vaucason - maszyna tkacka sterowana kartami perforowanymi
1801 - krosno J. Jaquarda sterowane kartami perforowanymi

1812(22) - **Ch. Babbage** (1792-1871) maszyna różnicowa (obliczanie tablic metoda interpolacji); 1833 maszyna analityczna - uznawana za prototyp współczesnych komputerów.



sterowanie sekwencyjne za pomocą kart perforowanych
pamięć (magazyn) i jednostka licząca (młyn)
wprowadzanie danych za pomocą kart perforowanych lub z tarcz numerycznych



1857 - arytmometr klawiszowy (USA)

1889 - **Hollerith** - zastosowanie kodowania binarnego na kartach dziurkowanych w czasie spisu narodowego
1889 - f-ma Burroughs - pierwsze urządzenie drukujące

1920 - sumator 4-działaniowy elektryczny .

1933 - **Zuse** (Niemcy) - automat liczący, przekaźniki, system dwójkowy (maszyna Z1)
- wprowadził "przedstawienie półlogarytmiczne" liczb czyli "postać zmiennopozycyjną"

1944 - **G. Stibitz** z Bell Tel. Lab. (USA) - Mark 1 i Bell-V: maszyny liczące na przekaźnikach, wprowadzanie danych i programu taśma perforowaną

1939-1946 - **Eckert i Mauchly**, Univ. Pensylwania (USA) - **Eniac**:
całkowicie elektroniczna, 18000 lamp
zewnętrzne sterowanie sekwencyjne (taśmy i karty perforowane, tablice połączeń)
pamięć tylko dla przechowywania danych
system dziesiętny.

1945 - John von Neuman (1903-1957)
1. zasada sterowania wewnętrznego: rozkazy i dane we wspólnej pamięci
2. system dwójkowy

1949 - EDVAC - realizacja pomysłów von Neumanna
1951- UNIVAC - pierwsza maszyna produkowana "masowo", pamięć rtęciowa i taśmowa
1953 - IBM 701 - system dwójkowy, pamięć elektrostatyczna (lampy oscyloskopowe)
1958 - Sietuń (ZSRR) - maszyna trójkowa
1960 – PDP-1 – pierwsza maszyna komercyjna z klawiaturą i monitorem jako I/O

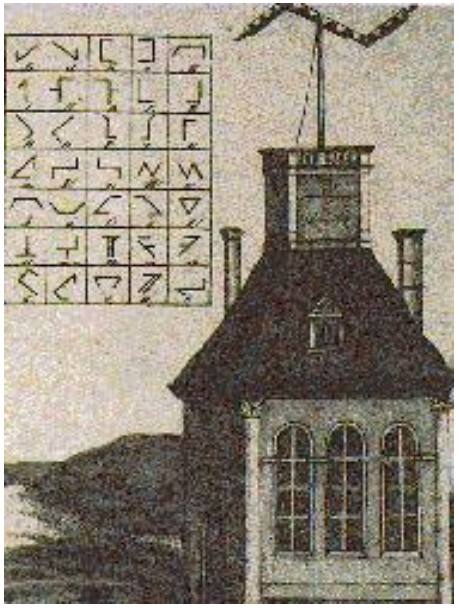
1962 – pierwsza gra komputerowa (Steve Russell)
1973 – eksperymentalny komputer PC (Xerox Alto)

1975 – pierwszy komercyjny PC Altair 8800
1981 – pierwszy IBM PC XT

Maszyny cyfrowe w Polsce:

Instytut Matematyki PAN - Grupa Aparatów Matematycznych
1950 - GAM-1 - przekaźnikowy
1958 - XYZ (częściowo kopia IMB 701)
1959 - powstaje ELWRO
1960/61 - ODRA 1001 (czyli UMC-1 Z. Pawlaka z 1958)
1964 - ODRA 1003 (tranzystorowa) - produkcja seryjna
1967 - ODRA 1300

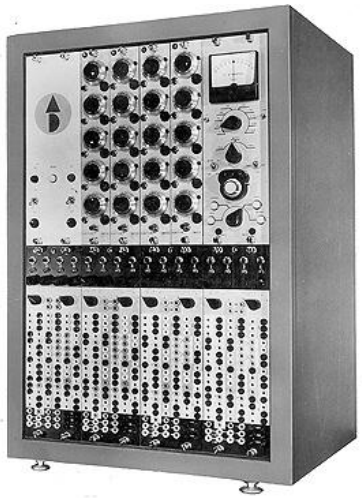
Przesyłanie sygnałów:



1791 - Claud Chappe - telegraf optyczny (15min/1000km/znak)
1794 - Linia Paryż-Lille
1825 - telegraf Schillinga z kodem dwójkowym
1837 - kod cyfrowy Morse'a
1841 - taśma perforowana w telegrafie (Wheatstone)
1851 - współczesny kod Morse'a
1999 - rezygnacja z kodu Morse'a, 40 Gb/s w światłowodzie.

Podział maszyn cyfrowych:

- analogowe (funkcje ciągłe, pierwowzór: suwak logarytmiczny)
- cyfrowe (wartości dyskretne, pierwowzór: liczydło)
- hybrydowe



Komputer analogowy - pomiędzy wartościami poszczególnych zmiennych istnieją ściśle określone związki sformułowane w postaci realizowanych sprzętowo zależności matematycznych. Zmiany wartości funkcji f(t) odwzorowywane są w nim za pomocą zmian ciągłych zmiennych fizycznych, np. napięcia lub ciśnienia. Komputer analogowy zbudowany jest z członów operacyjnych, które sprzętowo realizują konkretne równania matematyczne (np. układy hydrauliczne, elektryczne, pneumatyczne lub optyczne). Typ zastosowanych układów analogowych określa typ komputera (np. elektryczny, hydrauliczny).

Komputer cyfrowy

Pamięć operacyjna | **Sterowanie**

PROCESOR

Pamięć stała | **Wejścia/wyjścia**

Procesor – urządzenie elektroniczne albo abstrakcyjne urządzenie scharakteryzowane przez zbiór jego operacji elementarnych, tj. operacji, które mogą być wykonane przez procesor.

Wejście/wyjście - urządzenia dokonujące zamiany pomiędzy wewnętrzną reprezentacją informacji w komputerze a wielkościami (zawsze analogowymi) reprezentującymi te same informacje poza komputerem (np. średniowieczny manuskrypt lub sygnał cyfrowy).

Klasyczny **cykl von Neumanna** wykonania programu wynikowego:

1. pobranie rozkazu

2. ustawienie licznika rozkazów

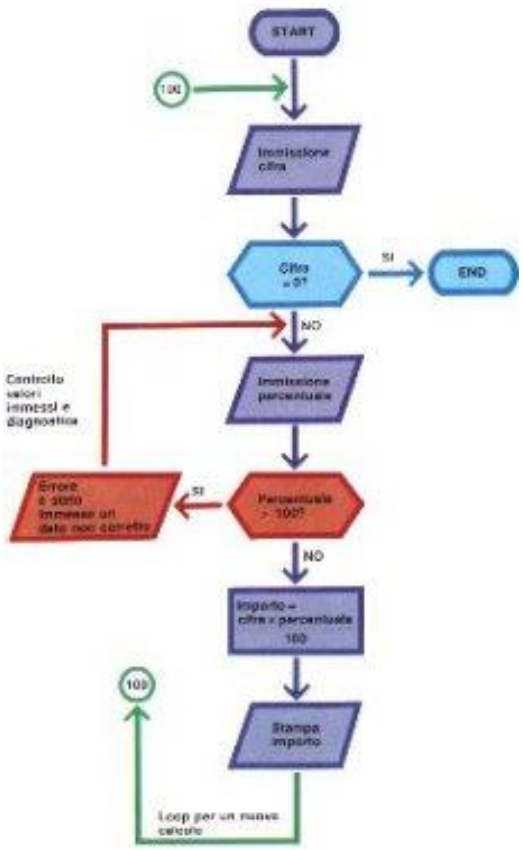
3. analiza rozkazu (wyznaczenie typu rozkazu i ew. adresów niezbędnych danych)

4. pobranie niezbędnych danych

5. wykonanie rozkazu

6. zapis wyniku w pamięci

7. skok do 1



Algorytm - metoda rozwiązania zadanego problemu w **skończonej ilości** operacji elementarnych (kroków algorytmu).

Język sformalizowany - język sztuczny, utworzony droga przyjęcia pewnej liczby aksjomatów i definicji. Charakteryzuje się: alfabetem, zbiorem słów (tzw. słów kluczowych), ściśle określonymi zasadami składni i semantyki (zasad określania znaczenia wyrażeń).

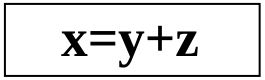
Język programowania - sformalizowany język służący do zapisu algorytmu.

- Pascal
- C
- C++
- Fortran90...

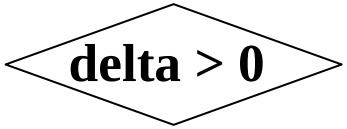
Algorytm najlepiej zapisać za pomocą **schematu blokowego**:

Schemat blokowy to graficzne przedstawienie ciągu kroków algorytmu. Sposób i kolejność działań programu określane są przez wzajemny układ i sposób połączenia bloków.

Początek lub koniec algorytmu. W każdym algorytmie musi być dokładnie jeden *start* (jedno wyjście) i jeden *stop* (dowolnie dużo wejść).

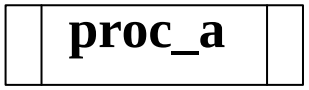


Blok instrukcji wykonywalnych (obliczenia i podstawienia). Musi mieć jedno wejście i jedno wyjście.

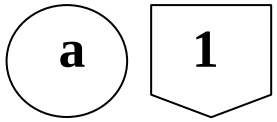


Decyzja logiczna (wybór warunkowy, rozgałęzienie algorytmu). Musi mieć dokładnie jedno wejście i dwa wyjścia. Decyzja logiczna zawsze powoduje konieczność rozważenia dwóch gałęzi algorytmu: gdy warunek jest spełniony i gdy nie jest.

Operacje wejścia/wyjścia. Musi mieć jedno wejście i jedno wyjście.



Blok instrukcji wcześniej zdefiniowanych (najczęściej wcześniej zdefiniowany podprogram). Musi mieć jedno wejście i jedno wyjście.



Łączniki stronicowe i międzystronicowe.

Jednokierunkowy łącznik elementów algorytmu. Wyznacza ścieżkę przebiegu algorytmu czyli *ścieżkę sterowania*.

Punkt połączenia gałęzi algorytmu.

1. Schemat blokowy jest *metajęzykiem*, dlatego w zasadzie powinien być zapisany w taki sposób, by był niezależny od *języka programowania* (powinien odpowiadać ciągowi operacji matematycznych i powinien być zapisany przy użyciu zwykłych operatorów matem.). Pewne operacje można zapisywać za pomocą *pseudokodu*.

2. Schemat musi być prosty i zrozumiały. W razie dużej złożoności algorytmu schemat należy podzielić go na części (najczęściej podprogramy – jest to naturalna droga do ***programowania strukturalnego***). Podprogramy rozrysowywane są z zasady na osobnych arkuszach i przyłączone za pomocą *symbolu bloku wcześniej zdefiniowanego*.

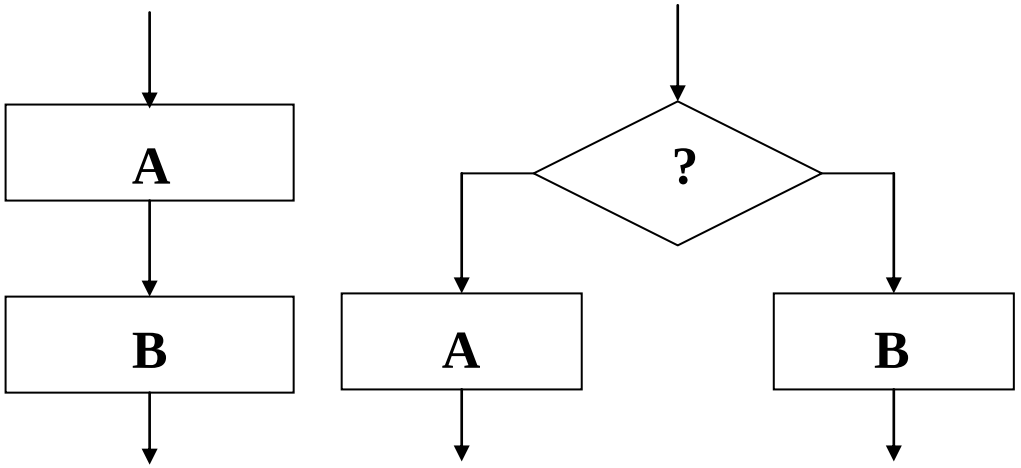
3. Bloki algorytmu zaleca się opatrywać krótkimi komentarzami.

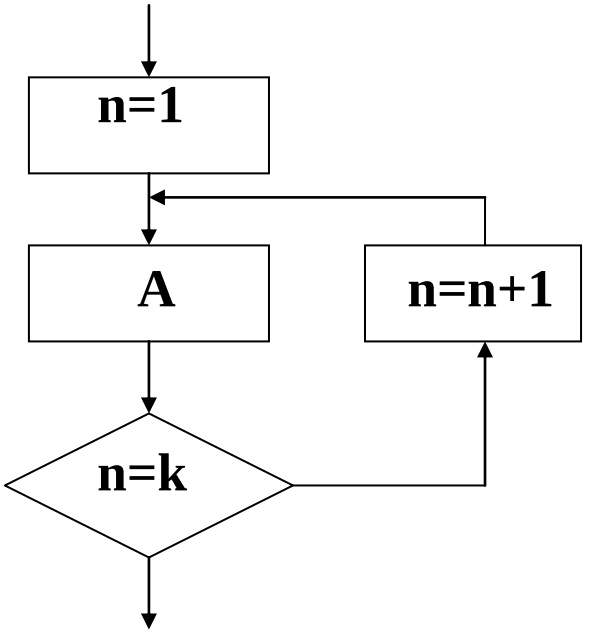
4. W jednym bloku instrukcji nie zaleca się umieszczania dużej liczby operacji.

5. Należy unikać rysowania przecinających się łączników elementów algorytmu. W razie konieczności należy zastosować łączniki stronicowe i międzystronicowe.

6. Schemat blokowy praktycznie zawsze jest wielokrotnie poprawiany, przerabiany i uzupełniany. Dlatego, kreśląc schemat blokowy należy zostawić miejsce na dokonywanie ew. korekt.

Następstwo instrukcji: **Decyzja logiczna (wybór warunkowy):** **Iteracja k-krotna:**

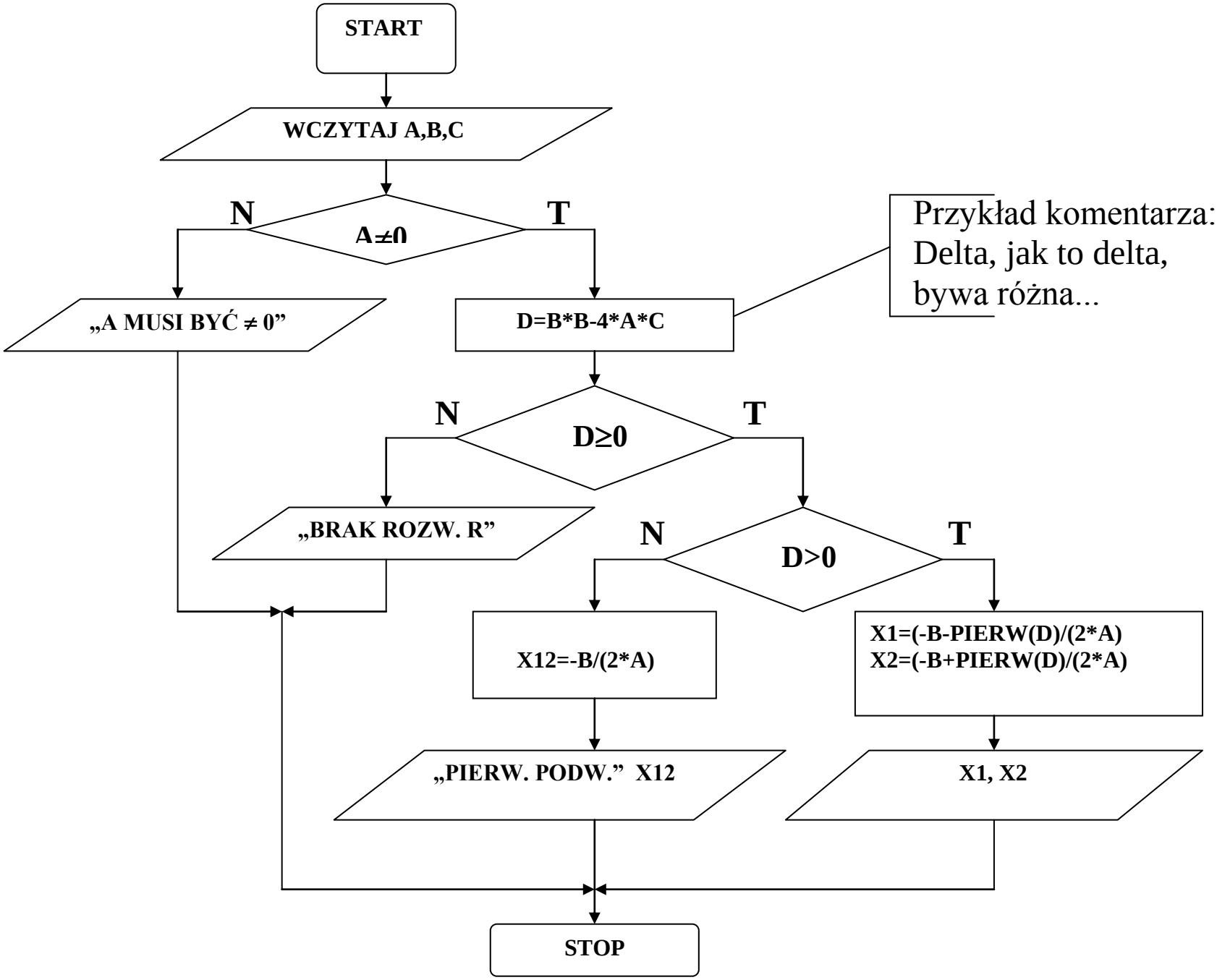




Rozpatrzmy równanie kwadratowe:
$y=ax2+bx+c \qquad a\neq 0$

$\Delta=b^2-4ac$

$\Delta>0 \quad x_1=(-b-\sqrt{\Delta})/(2a) \quad x_2=(-b+\sqrt{\Delta})/(2a)$
$\Delta=0 \quad x_{1,2}=-b/(2a)$
$\Delta<0 \quad$ ----


START
WCZYTAJ A,B,C
N
A≠0
T
„A MUSI BYĆ ≠ 0”
D=B*B-4*A*C
Przykład komentarza:
Delta, jak to delta,
bywa różna...
N
D≥0
T
„BRAK ROZW. R”
N
D>0
T
X12=-B/(2*A)
X1=(-B-PIERW(D)/(2*A)
X2=(-B+PIERW(D)/(2*A)
„PIERW. PODW.” X12
X1, X2
STOP

SPOSOBY OPRACOWANIA ALGORYTMÓW KOMPUTEROWYCH

- STRUKTURALNY – algorytm podzielony jest szereg procedur, realizujących wydzielone kroki rozwiązania zagadnienia. Możliwe jest tworzenie bibliotek gotowych procedur;

- OBIEKTOWY – procedury i dane tworzą obiekty reprezentujące najważniejsze elementy algorytmu;

- LINIOWY (JEDNOWĄTKOWY)– kolejne kroki algorytmu wykonywane są według kolejności ich wywołania, w danym momencie realizowana jest tylko jedna procedura;

- RÓWNOLEGŁY – wiele procedur wykonywanych jest jednocześnie, wymieniają się one danymi.

ALGORYTM ZSTĘPUJĄCY – podział problem na podproblemy, kolejny podział podproblemów na mniejsze zagadnienia itd., aż ich elementarnych kroków algorytmu.

PROBLEM 1:
JAK NAPISAĆ PROGRAM ?

Problem (zadanie)←świadomość & wiedza

↓

Algorytm←wiedza

↓

Język programowania←narzędzie

↓

Program źródłowy

Program źródłowy - zapis algorytmu w języku programowania.

Program wynikowy (wykonywalny) jest to ciąg operacji opisanych przez program źródłowy i przedstawiony w postaci ciągu instrukcji wykonywalnych przez określony komputer.

PROBLEM 2: JAK ZBROBIĆ PROGRAM WYKONYWALNY?

Program źródłowy

↓

Translacja (kompilator lub asembler)

↓

Konsolidacja

↓

Program wynikowy

PROBLEM 3: TESTOWANIE PROGRAMU

POZYCYJNE SYSTEMY LICZBOWE

System zapisu liczb: zbiór cyfr i prawideł przedstawiania liczb (np.: 0,1,...,9 albo X,V,C,I,L,M,D)

Systemy: **pozycyjne** i **nie-pozycyjne** (np. rzymski) 1978 = MCMDXXVIII

Każdą liczbę X można przedstawić jako sumę:

$$X= a_k p^k+a_{k-1}p^{k-1}+...+a_1p^1+a_0p^0+a_{-1}p^{-1}+a_{-2}p^{-2}+...+a_{-s}p^{-s}$$

gdzie p>1 i p∈N jest **podstawą systemu**, współczynniki a_k (**cyfry**) mogą przyjmować wartości od 0 do p-1.

Np. w dziesiątkowym systemie pozycyjnym (podstawa 10): $1978 = 1*10^3 + 9*10^2 + 7*10^1 + 8*10^0$

$a_3=1$, $a_2=9$, $a_1=7$, $a_0=8$

podstawa	Cyfry
2	0 1
8	0 1 2 3 4 5 6 7
10	0 1 2 3 4 5 6 7 8 9
16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Dowolna liczbę X w systemie pozycyjnym o podstawie p zapisujemy jako:

$$X=(a_k a_{k-1}...a_1 a_0.a_{-1}a_{-2}...a_{-s})_p$$

np. $x=(10001.001)_2$; $x=(A0F)_{16}$; **x=123.567**

10	**2**	**8**	**16**
256	100000000	400	100
3236	110010100100	6244	CA4
1256789222	1001010111010010001010011100110	11272212346	4AE914E6

OPERACJE ARYTMETYCZNE W POZYCYJNYCH SYSTEMACH LICZBOWYCH

4 podstawowe działania = tabliczki mnożenia i dodawania.

System dwójkowy (mamy do dyspozycji cyfry 0 i 1)

+	0	1
0	0	1
1	1	10

*	0	1
0	0	0
1	0	1

Uwaga: zapis 10 oznacza $1*2^1 + 0*2^0$ (czyli 2+0=2) a nie żadne „dziesięć”

System czwórkowy (mamy do dyspozycji cyfry 0, 1, 2 i 3)

+	0	1	2	3
0	0	1	2	3
1	1	2	3	10
2	2	3	10	11
3	3	10	11	12

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	10	12
3	0	3	12	21

Uwaga: zapis 10 oznacza $1*4^1 + 0*4^0$ (czyli 4+0=4) a nie żadne „dziesięć”

zapis 12 oznacza $1*4^1 + 2*4^0$ (czyli 4+2=6) a nie żadne „dwanaście”

Należy zapamiętać, że $2^N = 2^{N-1}+...+2\bullet 2^0$

Odejmowanie liczb binarnych: weźmy liczbę osiem

system "pożyczek": 1 0 0 0 ($1*2^3+0*2^2+0*2^1+0*2^0$=8 dec)

⇓ ⇓ ⇓ ⇓

0 1 1 10 ($0*2^3+1*2^2+$**$1*2^1+ 1*2^1$**$+0*2^0$=8 dec)

["Pożyczka" do 0 daje 10 czyli 2 dec]

np.; 10000000 (128 dec) 11111110

-1111111 (127 dec) $\Rightarrow$ - 1111111

= 1 [10-1=1 (zapis dwójkowy)]

	101101.1	
11101.	01	
101		11011.11
+1101.	-	*
011	11010.01	10.1
11011.		1101111
000	10011.01	
	1	
		+
		1101111
		1000101.
		011

LICZBY O SKOŃCZONEJ PRECYZJI

Wielkość komórki pamięci jest zdefiniowana przez hardware lub software

⇓

Liczby można wyrażać tylko za pomocą skończonych ciągów cyfr.

LICZBY STAŁOPOZYCYJNE

K bitów		
znak (1 bit)	część całkowita (L bitów)	część ułamkowa (K-L-1 bitów)

Znak: 0 dla liczb>0, 1 dla liczb<0.

Liczba największa: **1....1.1....1** = 1...1.1...1 + 0.0...1 - 0.0...1 = 10...0.0 - 0.0...1 = $2^{L}-2^{-(K-L-1)}$

Ilość znaków: L K L K L+1

Np. 111.111_2 (7 i 7/8) = (7 i 7/8) + 1/8 –1/8 = $1000_2 - 0.001_2 = 2^3 - 2^{-3} = 2^L - 2^{(7-3-1)}$

Liczba najmniejsza**: 0.0...1** = $2^{-(K-L-1)}$

Np. K=13 L=6 $1/64<x<63^{63}/_{64}$

Uwaga: występują błędy nadmiaru i niedomiaru.

LICZBY ZMIENNOPOZYCYJNE

Liczbę rzeczywistą można przedstawić jako: $X=f*10^e$ gdzie f nazywamy **mantysą,** f<1, e nazywamy **cechą.**

K bitów			
znak liczby (1 bit)	mantysa (R bitów)	znak cechy (1 bit)	cecha (K-R-2 bitów)

Zakres zapisu określa liczba cyfr cechy

Precyzję zapisu określa liczba cyfr mantysy

Liczba największa: $0.1.....1*10^{1...1} = (1.0 - 0.0....1)*10^{1...1} = (1 - 2^{-R})*2^{2^{(K-R-2)}-1}$

Liczba najmniejsza: $0.0...1*10^{-1...1} = 2^{-2^{(K-R-2)}}$

Np.: K=13 R=8 1/65536<x<32512

Liczby znormalizowane - liczby, w których pierwszą cyfrą mantysy jest 1. Algebra liczb znormalizowanych jest zamknięta ze względu na 4 działania podstawowe.

Np. cecha 2-cyfrowa, mantysa 3-cyfrowa 0.1<f<1 lub 0. Liczby dodatnie: 0.1-99 - .999e99 => 199 rzędów wielkości, 179100 liczb

nadmiar ujemny	wyrażalne liczby ujemne	ujemny niedomiar	dodatni niedomiar	wyrażalne liczby dodatnie	nadmiar dodatni

------ -10^{100} -10^{-100}----------0----------10^{-100} 100^{99} -------

Liczby zmiennopozycyjne nie tworzą kontinuum.

W każdym zbiorze liczb zmiennopozycyjnych o skończonej precyzji jest tylko skończona ilość liczb, co powoduje błędy zaokrąglenia.

Zbiór liczb o skończonej precyzji nie jest zamknięty ze względu na wszystkie 4 działania podstawowe:

1. występują błędy nadmiaru lub niedomiaru.
2. występują błędy nie należenia do zbioru.

Np.: Z=(000,...,999) ad1: 600+600=1200 (błąd nadmiaru) ad 2: 7/2 nie należy do Z.

Komputer działając poprawnie może dawać wyniki błędne z punktu widzenia matematyki klasycznej.

Cyfry znaczące - w zapisie dziesiętnym danej liczby wszystkie cyfry z wyjątkiem początkowych zer:

Zero jest cyfrą znaczącą tylko w przypadku, gdy:

- znajduje się między dwiema cyframi nie będącymi zerami, albo

- znajduje się na dowolnym miejscu po cyfrze nie będącej zerem, ale zawartej w liczbie z przecinkiem.

15.00245 - 7 cyfr znaczących; 0.000045 - 2 c.z.

$5.23 \bullet 10^{12}$ — 3 c.z. $5.2300 \bullet 10^{13}$ — 4 c.z.

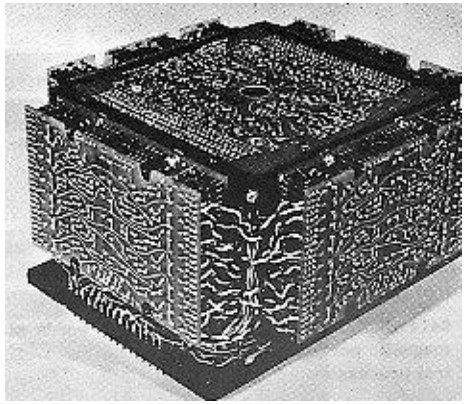
$500 = 5 \bullet 10^{2}$ – 1 c.z. $500 = 5{,}00 \bullet 10^{2}$ – 3 c.z.

0.25 – 2 c.z.. 0.2500 – 4 c.z..

0.25 – 2 c.z. 0.0025 – 2 c.z.

Do cyfr znaczących zalicza się również te zera końcowe, które nie wynikają z zaokrąglenia, lecz z rachunku:

1200 - 4 cyfry znaczące przy dokł. obliczeń 0.5; 1200 - 2 cyfry znaczące przy dokł. 50



Organizacja pamięci operacyjnej w komputerach cyfrowych**:**

- bit (ang. binary digit) : 0 lub 1
- 8 bitów = 1 byte (bajt)

1024 bity = 1 kb
1024 byte = 1 kB
1024 kB = 1 MB
1024 MB = 1 GB

- podział pamięci na jednostki (komórki, słowa) N-bitowe (najczęściej N=n*8) $\Rightarrow$ 2^N kombinacji bitów
- indywidualne adresy **N-bitowym** adresem $\Rightarrow$ **2^N adresów**

N	**2^N**
8	256
16	65536 = 64 kB
32	4 294 967 296 = 4GB
64	1.72e10 GB

Każda komórka pamięci może zawierać dowolną kombinację 0 i 1,
która może być zinterpretowana jako liczba, znak (litera, cyfra) lub rozkaz !!!

ZMIENNE i STAŁE

- Przez **zmienną/stałą** rozumie się **obszar pamięci wraz z zapisaną w niej informacją**.
- Fizycznie zmienna/stała wyznaczona jest przez **adres i rozmiar obszaru w pamięci o dostępie swobodnym RAM**.
- Informacji można nadawać różne znaczenia, nazywane **wartością zmiennej/stałej**.
- **Funkcję przyporządkowującą wartość konkretnemu obszarowi pamięci czyli sposób zinterpretowania ciągu bitów nazywamy typem zmiennej/stałej.**
- Każda zmienna/stała ma w każdej chwili dokładnie jeden typ, ustalany podczas procesu obliczeniowego.

⇓Adres

	11001111			
	00110011	**00001111**		
	00110011			
	11100111			

Zmienne/stałe są identyfikowane poprzez nazwy czyli ADRES

Typ zmiennej/stałej musi być określony przez programistę.

ALFABET FORTRANU

Niemal wszystkie drukowalne znaki kodu ASCII, tj.:

litery:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
• cyfry:
0 1 2 3 4 5 6 7 8 9
• znaki specjalne:
<spacja> _ = + - * / ( ) , . ' : ! " % & ; < > ? $
(znaki alfanumeryczne: litery, cyfry, _ *(podkreślenie))*

Ze znaków alfabetu tworzy się kod źródłowy, a w szczególności:
- Tworzy się nazwy obiektów (zmiennej, tablicy, procedury, modułu etc.);
- Zapisuje słowa kluczowe, operatory, instrukcje języka F90.

Znaki specjalne używane są jako **symbole operatorów**, **oznaczeń** oraz **do grupowania elementów**.
- Spacja służy w szczególności do rozdzielania elementarnych jednostek informacji F90 w instrukcjach
- Kilka spacji jest równoważne jednej spacji

Spacje mają znaczenie dla treści instrukcji
Użycie dużych i małych liter nie ma znaczenia dla treści instrukcji (za wyjątkiem stałych znakowych).

NAZWY OBIEKTÓW

Każdy z obiektów występujących w programie (zmienna, tablica, funkcja, procedura) identyfikowany jest poprzez nadaną mu nazwę.

- Nazwa składa się z ciągu liter, cyfr i znaków _
- Nazwa musi zaczynać się od litery
- Długość nazwy do 31 znaków
- Wielkość liter jest ignorowana (użycie dużych i małych liter nie ma znaczenia dla znaczenia nazwy)

prawidłowe nazwy:	**ala_ma kota**	**NaZwA**	**masa_galaktyki**
błędne nazwy:	**1dwatrzy**	**_start_**	**masa_gala ktyki**

Typy nazw: **globalne** i **lokalne**.

Nazwy globalne: identyfikowane (rozpoznawane) **w całym programie**. Są to nazwy procedur, funkcji, bloków common.

Nazwy lokalne: rozpoznawane są tylko **w ramach segmentu**, w którym są zadeklarowane. Są to nazwy zmiennych, tablic, argumentów i funkcji lokalnych.

SŁOWA KLUCZOWE

Zbiór (niewielki) słów, które mają ściśle zdefiniowane znaczenie (np. **STOP**, **DO** itd.).Z reguły są poleceniami wykonania pewnej operacji (np. zadeklarowania zmiennej czy wykonania skoku), służą do deklaracji zmiennych i stałych itp.

Słowa kluczowe nie są zarezerwowane, **ale zaleca się nie używać słów kluczowych dla oznaczania innych obiektów**.

Przykłady: **DO, STOP, CASE, IF**

OPERATORY

Operatory:

=	przypisania (podstawienie)
+, -, /, *	cztery działania podstawowe
< > ...	operatory relacji
.and. .or. ...	operatory logiczne

FORMAT KODU ŹRÓDŁOWEGO FORTRANU

a) Program (kod źródłowy) w języku Fortran składa się z **SEGMENTÓW**. Segmenty składają się z **instrukcji** oraz **komentarzy**.

b) **W każdym programie jest zawsze segment główny** i ewentualnie są **inne segmenty pomocnicze**.

c) Segmenty pomocnicze opisują **funkcje** i **procedury** zdefiniowane przez programistę (jak również na jeden ze sposobów pozwalają przypisywać wartości danym).

Segment główny musi rozpoczynać się od instrukcji PROGRAM:

PROGRAM *nazwa_programu*

Instrukcja ta identyfikuje dany segment jako główny i nadaje programowi indywidualną nazwę.

Uwaga: nazwa_programu jest nazwą globalną w ramach całego kodu wykonywalnego i musi być nazwą unikatową.

Np.: **program FOO**

Segment główny programu źródłowego musi kończyć się:

a) instrukcją **STOP**, powodującą bezwarunkowe zatrzymanie wykonania programu:

STOP

b) instrukcją **END PROGRAM** oznaczającą koniec segmentu głównego:

END PROGRAM *nazwa_programu*

KOLEJNOŚĆ INSTRUKCJI W SEGMENCIE

Fortran wymaga zapisywania instrukcji w ściśle zdefiniowanej kolejności.

Instrukcje o tej samej wymaganej kolejności wykonywane są w kolejności zadanej przez programistę poprzez:

- uszeregowanie w programie (od początku do końca):
- zastosowanie instrukcji sterujących kolejnością wykonania innych instrukcji.

W pierwszym przybliżeniu można powiedzieć, że obowiązuje następującą kolejność instrukcji w segmencie:

1. nagłówek segmentu	**PROGRAM foo**	(tu jedynie segment główny)
2. deklaracje	**IMPLICIT NONE**	(obow. w ELF90)
	REAL :: I,J	(zmienne nie wykorzystane!)
3. instrukcje "wykonywane" (obliczenia i "wejścia/wyjścia")	**write (*,*) 'Hello, I am HAL 9000'**	
4. instrukcja końca segmentu	**STOP**	
	END PROGRAM foo	

INSTRUKCJE

Instrukcja pozwala określić (*nakazać*) obliczenia lub operacje jakie ma przeprowadzić komputer.

STAŁE, **SŁOWA KLUCZOWE**, **NAZWY** to „słowa" języka Fortran

„Słowa" można łączyć w => **WYRAŻENIA**

WYRAŻENIA można łączyć w => **INSTRUKCJE**

POSTAĆ WYRAŻEŃ FORTRANU

Def.: operand to stała, zmienna, funkcja lub wyrażenie(!)

operand operator_dualny operand	**x+y**
operator_unitarny operand	**-y**
operand operator operand operator operand	**x+5*y**
operand operator (operand operator operand)	**x+(5*y)**

ZASADA WYZNACZANIA WARTOŚCI WYRAŻENIA (KOLEJNOŚĆ ANALIZY)

najpierw **sub-wyrażenia** ograniczone parą nawiasów ()

↓ **sub-wyrażenia** z operatorami unitarnymi

kolejno sub-wyrażenia powiązane operatorami o malejących wagach

na końcu **od lewej do prawej** dla operatorów o tej samej wadze

WAGI OPERATORÓW

najwyższa

najniższa

- negacja
****** potęgowanie
*** /** mnożenie i dzielenie
+_ dodawanie i odejmowanie
// łączenie łańcuchów
.eq. .ne. .lt. .le. .gt. .ge. relacje
== /= < <= > >=
.not. negacja logiczna
.and. suma log.
.or. alternatywa log.
.eqv. .neqv. równoważność lub nierównoważność

Uwagi: x^{-y} zapisujemy **x**(-y)** zaś **x**y**z** jest obliczane jako: **x**(y**z)**

RODZAJE INSTRUKCJI FORTRANU

1. **Instrukcje bierne (deklaracje)**, które opisują i deklarują obiekty, na których wykonywane będą operacje.

```
real :: i, j, k, pi
```

2. **Instrukcje przypisania** (czyli nadania obiektowi wartości);

```
b=3.14159*a*a
```

3. **Instrukcje wejście/wyjścia** (wejście: "dane", wyjście: "wyniki");

```
write (*,*) ‘Hello, I am Hal 9000’
```

4. **Instrukcje sterujące** (sterują przebiegiem wykonania programu).

```
do i=1,100
      j=i*5.12
end do
```

FORMAT INSTRUKCJI KODU ŹRÓDŁOWEGO ELF90

I. Zmienne, stałe, etykiety w instrukcjach muszą być oddzielone od sąsiednich słów kluczowych, zmiennych, stałych lub etykiet za pomocą jednej lub więcej spacji lub przez koniec instrukcji.

```
      program□foo
      implicit□none
      real :: a=2.0, b=3.0, c=4.0 ,d=5.0
      write (*,777) a,b,c,d
777□ format (' a=',f5.2,' b=',f5.2,' c=',f5.2,' d=',f5.2)
```

□ - tak wyjątkowo oznaczyliśmy w przykładzie spację „konieczną”

Sąsiednie słowa kluczowe muszą być rozdzielone spacją (z pewnymi wyjątkami, w szczególności dla słów kluczowych **end** oraz **else**)

Np. **end do** ⇔ **enddo**
else if ⇔ **elseif**

II. Instrukcje zapisuje się w liniach liczących do 132 znaków

```
a=12.345*b*c + 1.23 * ( a + 3.0*b + d*ef/g) – 3.14159
                b=2.*a
```

Przypomnienie: spacje oddzielają poszczególne elementy kodu i są znaczące.

III. Instrukcja może być zapisana w max. 40 liniach, W TYM w max. 39 liniach kontynuacji. Każda linia instrukcji, która jest kontynuowana w następnej linii, musi być zakończona znakiem &:

```
X = ((-2.0*a+1.7*b)*c)  &
    + (D+2.345)*e       &
    / (2.5*e)
      X = ((-2.0*a+1.7*b)*c)  &
&           + (D+2.345)*e       &
&           / (2.5*e)
```

Jeśli stała znakowa, nazwa zmiennej, słowo kluczowe lub etykieta są rozdzielone na dwie linie, to pierwszym znakiem nowej linii różnym od spacji musi być znak &. **Nie zaleca się dzielenia!!!**

```
program foo
implicit none
integer :: i,karamba      ! deklaracja zmiennych
 karamba=20
 i=kara&                  ! linia 1 instrukcji
   &mba                   ! linia 2 instrukcji
 write(*,*) i
stop
end program foo
```

```
program foo_ERROR
implicit none
integer :: i,karamba
 karamba=20
 i=kara &  ! linia 1
   & mba   ! linia 2
 write(*,*) i
stop
end program foo_ERROR
```

IV. W F90/95 etc. można w jednej linii zapisać wiele instrukcji rozdzielonych znakiem **;**

```
A = 0; b = 0; c=0;
```

Oprócz zapisu trywialnych instrukcji nie zaleca się takiego stylu programowania.

W ELF90 w linii można zapisać tylko jedną instrukcję.

V. Znak ! oznacza początek komentarza (za wyjątkiem wystąpienia w stałej znakowej). Komentarz rozciąga się od znaku ! do końca linii. Komentarze są ignorowane przez kompilator.

```
dlug_polokr=3.14*prom      ! Dlugosc polokregu
```

Uwaga: każda linia zaczynająca się znakiem ! a także każda linia złożona z samych spacji lub pusta jest ignorowana przez kompilator. Ponieważ komentarz może zawierać dowolny znak (także &) więc linia komentarza nie może mieć kontynuacji

```
! to jest komentarz skdhfsfhosfh  oiefhosiefhr sliefjlsie wejrlswaoer wpoefjrwloe&
”wopejrwpoefjr (blad)
! a to jego druga linia jeśli trzeba napisać baaaaaardzo długi komentarz
```

Linie komentarza mogą być swobodnie wstawiane pomiędzy kolejne linie instrukcji:

```
X = ((-2.0*a+1.7*b)*c)  &
! bardzo długie i trudne wyrazenie arytmetyczne ☺
     + (D+2.345)*e        &
      / (2.5*e)
```

A więc aby kompilator zechciał zaakceptować nasz program, musimy zadbać, by:

1. Kod zapisany był w liniach liczących do 132 znaków.
2. Komentarze zaczynały się od znaku **!**
3. Spacje umieszczone były w odpowiednich miejscach
4. Linie instrukcje kontynuowane w kolejnym wierszu kończyły się znakiem **&** (z max. 39 liniami kontynuacji).
5. W linii można zapisać tylko jedną komendę (ELF90).
6. Segment główny musi rozpoczynać się od instrukcji **PROGRAM**.
7. W każdym segmencie musi zostać zastosowana instrukcja **IMPLICIT NONE** (ELF90).
8. Typ każdej zmiennej musi być explicite zadeklarowany poprzez deklarację typu (ELF90).
9. Segment główny programu wynikowego musi kończyć się inst. **STOP** i **END PROGRAM**.

```
PROGRAM foo
STOP
END PROGRAM foo
```

```
PROGRAM foo2
write (*,*) 'Hello, I am HAL9000'
STOP
END PROGRAM foo2
```

```
PROGRAM foo3
IMPLICIT NONE
REAL :: a
STOP
END PROGRAM foo3
```

ZMIENNE i STAŁE

- Fizycznie zmienna/stałą to **obszar pamięci komputera** z danym **rozkładem ładunków** (0 i 1)**.** Obszar ten wskazywany jest przez **adres** i ograniczony poprzez zdefiniowany **rozmiar**.

- Przez **zmienną/stałą** rozumie się **obszar pamięci wraz z zapisaną w niej informacją.** Informacji można nadawać różne znaczenia, nazywane **wartością zmiennej**.

Funkcję przyporządkowującą konkretne znaczenie konkretnemu obszarowi pamięci (zapisanej w nim informacji) czyli sposób zinterpretowania ciągu bitów nazywamy **TYPEM ZMIENNEJ**.

- Każda zmienna/stała ma w każdej chwili dokładnie jeden typ, ustalany podczas procesu obliczeniowego.

TYPY ZMIENNYCH i STAŁYCH

W F90 mamy **5 predefiniowanych TYPÓW** zmiennych i stałych:

INTEGER, REAL, COMPLEX, LOGICAL, CHARACTER

Dodatkowo oraz **można definiować własne TYPY POCHODNE**.

W FORTRANIE STOSOWANE SĄ DWIE METODY OKREŚLANIA TYPÓW ZMIENNYCH:

1. Typ zmiennej/stałej (i w ogólności każdego obiektu) w F90 określa się za pomocą **instrukcji deklaracji typu, określających dla każdej zmiennej/stałej jej: TYP, RODZAJ** i **ATRYBUTY. Deklaracja typu każdego obiektu jest obowiązkowa w ELF90** i jest zalecana do stałego użycia podczas programowania w Fortranie i każdym innym języku programowania!!!

2. Przy zastosowaniu tzw. konwencji znaków:

Zmienne/stałe o nazwach zaczynających się na litery a-h,o-z są typu REAL

Zmienne/stałe o nazwach zaczynających się na litery i-n są typu INTEGER

Jest to metoda nie zalecana. Jest to metoda szybka, często stosowana ale podatna na błędy.

Rodzaj czyli **KIND**

- Każda zmienna/stała w F90 ma zawsze określony **rodzaj** (*ang. kind*) – decyduje on o **ZAKRESIE WARTOŚCI I DOKŁADNOŚCI ZAPISU** zmiennej/stałej.

- Zmienne/stałe mogą mieć rodzaj (KIND) **określony jawnie** lub **ustalany domyślnie**.

- **Każdy typ predefiniowany zmiennej/stałej ma swój KIND predefiniowany** (tzw. domyślny), lecz można taką zmienną/stałą zadeklarować stosując inny rodzaj (KIND), zależnie od konkretnie: użytego sprzętowego procesora, cech kompilatora i potrzeb programisty.

- Każdy **rodzaj (KIND)** jest rozpoznawany i identyfikowany poprzez skojarzoną z nim **liczbę naturalną M** nazywaną **parametrem rodzaju**.

Np. jeśli dla liczb całkowitych parametr rodzaju M=4, to znaczy, że zakres wartości liczb całkowitych tego rodzaju wynosi od -2^{n-1} do $+2^{n-1}-1$ gdzie n=M*8, czyli –2 1474 83 648 do 2 147 483 647.

WARTOŚCI PARAMETROW RODZAJU DOPUSZCZALNE PRZEZ KOMPILATOR ELF90

Type	Kind Type Parameter	Notes
INTEGER	4*	Range: -2,147,483,647 to 2,147,483,647
REAL	4*	Range: $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ Precision: 7-8 decimal digits
REAL	8	Range: $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ Precision: 15-16 decimal digits
COMPLEX	4*	Range: $1.18 * 10^{-38}$ to $3.40 * 10^{38}$ Precision: 7-8 decimal digits
COMPLEX	8	Range: $2.23 * 10^{-308}$ to $1.79 * 10^{308}$ Precision: 15-16 decimal digits
LOGICAL	4*	Values: .TRUE. and .FALSE.
CHARACTER	1*	ASCII character set

* - oznacza wartość domyślną dla kompilatora ELF90

STAŁE LICZBOWE TYPU CAŁKOWITEGO

Łańcuch cyfr i ew. znak -/+, przedstawiających liczbę całkowitą z pewnego ograniczonego zakresu. W ELF90 KIND domyślny wynosi 4.

Słowo kluczowe	Zakres
INTEGER	**–2 1474 83 648 do 2 147 483 647**

Przykłady:
0
-999
+10

SPOSOBY DEFINIOWANIA PARAMETRU KIND STAŁYCH LICZBOWYCH TYPU CAŁKOWITEGO

a) poprzez podanie wartości parametru rodzaju ***explicite*** w zapisie stałej (sposób nie zalecany)

1.0_4
0.5_2 (nie działa w ELF90)
-999.23_4
+10.1_6 (nie działa w ELF90)

b) poprzez podanie wartości parametru rodzaju jako ***parametru*** w zapisie stałej

```
integer, parameter :: short=4
…..
-1234_short
2_short
```

c) poprzez wykorzystanie funkcji określającej automatycznie wymaganą wartość parametru rodzaju w zależności od potrzebnego ***przedziału wartości***.

Niech J jest takie, że liczba I jest z zakresu $-10^j < I < 10^j$.

```
integer, parameter :: k4=selected_int_kind(4)
….
-1234_k4
```

przykład:

```
PROGRAM FOO
IMPLICIT NONE                        ! Obowiazkowe wylaczenie konwencji znakow
INTEGER, PARAMETER :: K4=4           ! Dekl. K4 jako parametru typu całkowitego (=4)
A=1234_K4                            ! Przypisanie zmiennej a wartości 1234 (kind=4)
A=13_4                               ! Przypisanie zmiennej a wartości 13 (kind=4)
STOP
END PROGRAM FOO
```

ZMIENNE TYPU CAŁKOWITEGO

Zmienna całkowita to obszar pamięci zawierający informację (binarną), która jest interpretowana jako reprezentacja liczby całkowitej i określa wartość tej zmiennej.

Zmienna całkowita jest dokładną reprezentacją odpowiadającej jej liczby całkowitej.

Instrukcja **INTEGER** deklaruje zmienną jako zmienną typu całkowitego oraz określa jej cechy:

Składnia: **INTEGER [([kind=]N)] [,lista_atrybutów] :: nazwa_zmiennej [(specyfikacja_tablicy)] [=wartosc_poczatkowa] [,...] nazwa_funkcji**

(uwaga: to jest pojedyncza instrukcja)

N – stała całkowita lub wyrażenie skalarne typu całkowitego (musi być możliwe do wyliczenia w czasie kompilacji)

wartość_początkowa - wyrażenie skalarne typu całkowitego (musi być możliwe do wyliczenia w czasie kompilacji)

1. lista_atrybutów: **PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), INTENT (IN)/INTENT (OUT)/INTENT (INOUT), PUBLIC/PRIVATE, OPTIONAL, POINTER, SAVE, TARGET**.

2. Każdy z atrybutów może wystąpić tylko raz w danej instrukcji **INTEGER**
3. Jeśli użyto atrybutu **PARAMETER**, wartość początkowa musi być podana

Po co atrubut *parameter*?

- Jeśli zmienna może i musi zachować tylko jedną wartość. Każda próba (zamierzona lub niezamierzona) zmiany jej wartości spowoduje postanie błędu wykonania.
- Dla podkreślenia w programie za pomocą nazw symbolicznych specjalnych liczb, np. stałych fizycznych.
- Dla ew. ułatwienia zmian wartości tego parametru w całym segmencie programu.

```
program intdek
implicit none
integer :: a, b, c, d=12, e=12*2-1
integer (kind=4) :: f, g=24, h=2*5-1          ! integer (kind=2) błąd bo ELF90 nie dopuszcza kind=2
integer (4) :: i, o=22, p=2*8
integer, parameter :: k4=4, j=21
integer (kind=4), parameter :: l=5, n=2*4-1
integer (4), parameter :: s=4, t=3*6-n
stop
end program intdek
```

Instrukcja **IMPLICIT NONE**

Instrukcja powoduje **wykluczenie** konwencji znaków przy ustalaniu typu zmiennych.

Składnia: **IMPLICIT NONE**

W elf90 instrukcja **Implicit None MUSI** zostać umieszczona w każdym segmencie programu, w którym deklarowane są typy zmiennych.

```
program foo
implicit none
integer :: i,j          ! dekaracja zmiennych i j jako typu całkowitego
j=4
i=j+j
write(*,*) j,i
stop
end program foo
```

STAŁE LICZBOWE TYPU RZECZYWISTEGO

Postać: **znak -/+, łańcuch cyfr część całkowita, kropka dziesiętna, łańcuch cyfr część dziesiętna, ew. znak -/+ i wykładnik części wykładniczej**.
Taki ciąg znaków przedstawia liczbę rzeczywistą z pewnego, ograniczonego zakresu.
KIND domyślny w ELF90 to 4, dozwolony także KIND=8.

Słowo kluczowe	Zakres
REAL	**$1.18*10^{-38}$ do $3.4*10^{38}$**

Zakres wartości liczb rzeczywistych KIND=4:

-3.402..E+38 do **-1.175..E-34** oraz **1.175..E-34** do **3.402.. E+38**

Jeśli KIND=8, odpowiednio zakres= (±) **$2.23*10^{-308}$** do (±) **$1.79*10^{308}$**

Przykłady: stałe liczbowe typu rzeczywistego o rodzaju KIND=4:

1.0
-999.23
+10.11
10.23e-11 oznacza **$10.23*10^{-11}$**

SPOSOBY DEFINIOWANIA RODZAJU (PARAMETR KIND) STAŁYCH LICZBOWYCH TYPU RZECZYWISTEGO

a) poprzez podanie wartości rodzaju ***explicite*** w zapisie stałej (sposób nie zalecany)

1.0_4 **0.1_2** (nie w ELF90) **-999.11_4** **+10.2_6** (nie w ELF90)

b) poprzez podanie wartości rodzaju jako ***parametru*** w zapisie stałej

```
integer, parameter :: short=4
-1234.56_SHORT
2.1_ SHORT
```

c) wykorzystanie ***funkcji określającej automatycznie*** wymaganą wartość parametru KIND w zależności od potrzebnego przedziału wartości i precyzji obliczeń.

Niech liczba musi mieć co najmniej 9 cyfr znaczących i zakres wykładnika ±88.

```
integer, parameter :: long=selected_real_kind(9,88)
```

Notacja:

```
-1234.0_long
```

zapewnia nam zapis wartości stałej rzeczywistej w pamięci RAM z wymagana precyzją.

Przykład:

```
PROGRAM HALLO_DAVE
IMPLICIT NONE
real :: a                          ! Deklaracja a jako zmiennej typu rzeczywistego
integer, parameter :: short=4      ! Dekl. short jako parametru typu całkowitego (=4)
a=1234.0_short                     ! Przypisanie zmiennej a wartości 1234.0 (kind=4)
WRITE (*,*) 'a=',a
STOP
END PROGRAM HALLO_DAVE
```

Temat będzie bardziej szczegółowo omawiany później.

ZMIENNE TYPU RZECZYWISTEGO

Zmienna rzeczywista to obszar pamięci zawierający informację (binarną) która jest interpretowana jako reprezentacja liczby rzeczywistej i określa wartość tej zmiennej. Zmienna rzeczywista jest przybliżeniem odpowiadającej jej liczby rzeczywistej.

Instrukcja REAL deklaruje m.innymi zmienną jako zmienną typu rzeczywistego:

REAL [([kind=]N)] [,lista_atrybutów] :: nazwa_zmiennej [(specyfikacja_tablicy)]
[=wartosc_poczatkowa][,...] nazwa_funkcji

(uwaga: to jest pojedyncza instrukcja)

N – stała całkowita lub wyrażenie skalarne typu całkowitego (musi być możliwe do wyliczenia w czasie kompilacji);

wartość_początkowa - wyrażenie skalarne typu całkowitego (musi być możliwe do wyliczenia w czasie kompilacji)

1. lista_atrybutów: **PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), INTENT (IN)/INTENT (OUT)/INTENT (INOUT), PUBLIC/PRIVATE, OPTIONAL, POINTER, SAVE, TARGET.**
2. Każdy z atrybutów może wystąpić tylko raz w danej instrukcji REAL
3. Jeśli użyto atrybutu PARAMETER, wartość początkowa musi być podana.

```
program reldek
implicit none
real :: a, b, c                          ! a, b oraz c sa typu real zapisane na 4 bajtach
real (kind=8) :: f                       ! f jest zmienną typu całkowitego, kind=8 (8 bajtów)
                                         ! real (kind=2) :: j=5 błąd (ELF90 nie dopuszcza)
real, parameter :: d=21.0                ! d jest zmienną rzeczywistą o „zablokowanej” wartości 21
                                         ! domyślnie kind=4
real (kind=4), parameter :: g=5.0        ! d jest zmienną rzeczywistą o „zablokowanej” wartości 5,
                                         ! explicite kind=4
real :: e=12.0                           ! e jest zmienną typu rzeczywistego, przypisano jej wart. 12,
                                         ! implicite kind=4
f=5.0+2_4                                ! wyrażenie poprawne
                                         ! d=32 byloby wyrażeniem błędnym, próba przypisania
                                         ! wartości parametrowi
stop
end program reldek
```

PRZYPISANIE WARTOŚCI DO ZMIENNEJ SKALARNEJ

Postać ogólna instrukcji przypisania:

zmienna = wyrażenie

zmienna – zmienna skalarna
wyrażenie – wyrażenie arytmetyczne skalarne

Jeżeli typ wyniku wyznaczenia wyrażenia skalarnego nie jest taki sam jak typ zmiennej, automatycznie dokonywana jest konwersja typów

Typ zmiennej	przypisywany typ wyniku wyrażenia skalarnego
I	I(wyrażenie,kind(zmienna))
R	R(wyrażenie,kind(zmienna))
C	C(wyrażenie,kind=kind(zmienna))

a. przypisanie wart. rzeczywistej do zmiennej typu całkowitego

```
i=7.3                          ! wynik: i=7,   i jest typu INTEGER
```

b. przypisanie zmiennej o większej precyzji do zmiennej o mniejszej precyzji

```
r1=3.14159265234567  ! r1 kind=8
r2=r1                ! r2 kind=4,    wynik: utrata precyzji
```

c. przypisanie liczby typu complex do zmiennej typu real

```
a=(2.53, 4.12)          ! a typu REAL, wynik:  utrata części urojonej, a=2.53
```

d. przypisanie stałej do zmiennej

```
a = 1.7                 ! a real kind=8
b = 1.7_long            ! b real kind=8, większa precyzja
```

AUTOMATYCZNA KONWERSJA TYPÓW

Działania +, -, *, / na operandach a i b				
Typ a	**Typ b**	**Uzyty typ a**	**Użyty typ b**	**Typ wyniku**
I	I	a	b	I
I	R	R(a)	b	R
I	C	C(a)	b	C
R	I	a	R(b)	R
R	R	a	b	R
R	C	C(a)	b	C
C	I	a	C(b)	C
C	R	a	C(b)	C
C	C	a	b	C

Rezultat: **6/3=2 8/3=2 -8/3=-2 2/3=0**

Działanie ** na operandach a i b				
Typ a	Typ b	Użyty typ a	Użyty typ b	Typ wyniku
I	I	a	b	I
I	R	R(a)	b	R
I	C	C(a)	b	C
R	I	a	b	R
R	R	a	b	R
R	C	C(a)	b	C
C	I	a	b	C
C	R	a	C(b)	C
C	C	a	b	C

Rezultat: **2**3=8** **2**(-3)⇔1/(2**3)=0**

```
PROGRAM RI
IMPLICIT NONE
Real :: a, bn, pi=3.14159          ! zmienne a, bn sa typu real
Real, parameter :: pi=3.14159      ! parametr „pi” jest  typu real
Integer :: rr, jk                  ! zmienne rr, jk sa typu integer
rr=1
bn=pi*rr
rr=2.71                            ! uwaga na automatyczna konwersje typow
STOP
END PROGRAM RI
```

STAŁE TYPU ZNAKOWEGO (ŁAŃCUCHY)

Stała znakowa to obszar pamięci zawierający informację interpretowaną jako reprezentacja ciągu znaków pisarskich. Ciąg ten jest jednocześnie wartością tej zmiennej. Stała znakowa jest dokładną reprezentacją odpowiadającego jej ciągu znaków.

Słowo kluczowe	Długość ciągu znaków
CHARACTER (def. kind=1)	Od 0 do 32767

Stała znakowa zajmuje dokładnie tyle bajtów, ile wynosi jej długość (zerową jeśli użyjemy zapisu ‘’).

Stała znakowa składa się z ciągu znaków ASCII w postaci:

'znaki_ASCII' lub **„znaki_ASCII”** lub **ikind_’znaki_ASCII’**

gdzie **ikind** jest stałą typu integer (parametrem) o wartości 1.

‘gwiezdne wojny’
„wladca pierscieni” **nnn=1**
1_’das boot’ **nnn_’the_submarine’**

Znak ' w stałej znakowej uzyskuje się poprzez użycie ' ' : 'Mac''coy' ⇒ Mac'coy

---

Bardzo_dlugi_lancuch = ‘”Litwo, ojczyzno moja, &
&Ty jestes jak zdrowie;”’

W liniach kontynuacji stała znakowa musi **zaczynać się** znakiem **&**.

ZMIENNE ZNAKOWE (ŁAŃCUCHOWE)

Instrukcja **CHARACTER** deklaruje zmienną jako zmienną typu znakowego oraz określa jej cechy:

CHARACTER [selektor] [,lista_atrybutów] :: nazwa_zmiennej(specfikacja_tablicy)
[*char_len][=wartosc_poczatkowa][,...]
nazwa_funkcji(tablicy)[*char_len]

selektor długości

([len=]*) **(*** tylko dla argumentów aktualnych, stałych (parametrów) oraz funkcji zewnetrznych)

lub

([len=]int_exp) (wyliczalne w momencie przekazania sterowania do danego segmentu)

np.: **(len=*)** **(*)** **(len=12)** **(len=3*i)** **(5)**

CHARACTER (LEN=12) :: FILENAME

CHARACTER (LEN=*) :: COSIEWPISZE

selektor długości i kind'u:

([len=] int_exp, KIND= kind_par)

lub

(KIND= kind_par, LEN=int_exp)

Kind_par - wyrażenie skalarne typu całkowitego (musi być możliwe do wyliczenia w czasie kompilacji)

np.: **(len=12, kind=1)** **(len=2*naplen, kind=mykind)** **(12, kind=1)** **(kind=1, len=2*naplen)**

CHARACTER (LEN=12, KIND=1) :: FILENAME

CHARACTER (KIND=1) :: FILENAME*24

CHARACTER :: FILENAME*24='ale pokomplikowali…'

CHARACTER (LEN=22, KIND=1), parameter :: FILENAME='ale pokomplikowali… '

CHARACTER (*, KIND=1), parameter :: FILENAME='ale pokomplikowali… '

4. lista_atrybutów: PARAMETER, ALLOCATABLE, DIMENSION(*array-spec*), INTENT (IN)/INTENT (OUT)/INTENT (INOUT), PUBLIC/PRIVATE, OPTIONAL, POINTER, SAVE, TARGET.

5. Każdy z atrybutów może wystąpić tylko raz w danej instrukcji CHARACTER
6. Jeśli użyto atrybutu PARAMETER, wartość początkowa stałej znakowej musi być podana

OPERACJE NA ŁAŃCUCHACH

1. Kontaktacja łańcuchów czyli ich łączenie

Długość sumy łańcuchów (zmiennych znakowych) równa jest sumie długości łączonych łańcuchów (zmiennych znakowych).

np:
```
CHARACTER :: string1*8, string2*8
CHARACTER (len=16) :: string3,string4
string1='F90 jest'
string2=' cool'
string3=string1//string2                ! string3   'F90 jest cool    '
string4=string3//' i latwy w nauce'     ! string4  ' F90 jest cool    '
```

2. Fragmentacja łańcuchów

Niech STRING1 będzie typu **CHARACTER** (łańcuch). Wtedy fragment zmiennej wybiera się za pomocą wyrażenia:

STRING2=STRING1([pocz]:[kon])

STRING2=STRING1([pocz]:[kon])

gdzie:

pocz - wyrażenie arytmetyczne typu INTEGER podające numer pierwszej litery łańcucha STRING1, która wejdzie w skład łańcucha STRING2 (będącego zmienną znakową lub stałą znakową (łańcuchem)).

kon - wyrażenie arytmetyczne typu INTEGER podające numer ostatniej litery łańcucha STRING1, która wejdzie w skład łańcucha STRING2;

Prawidła:

- 1<=pocz<=kon<=length(STRING1)
- nie podanie pocz oznacza, ze pocz = 1;
- nie podanie kon oznacza, ze kon = długość łańcucha STRING1
- : oznacza cały łańcuch STRING1

np:
```
CHARACTER :: STRING1*32, STRING2*10
STRING1='ASTRONOMIA ROK PIERWSZY'
STRING2=STRING1(2:6)                    ! STRING2='STRON'
STRING2=STRING1(1:10)                   ! STRING2='ASTRONOMIA'
STRING2=STRING1(2:6)//STRING1(1:1)      ! STRING2='STRONA'
 STRING2=’ASTRONOMIA’(2:6)              ! STRING2=STRON’
```

INSTRUKCJA PĘTLI DO

Instrukcja DO (tzw. instrukcja pętli) powoduje wielokrotne (od 0 do wielu razy) wykonanie bloku instrukcji zawartych pomiędzy tą instrukcją DO i instrukcją END DO kończącą pętlę.

[nazwa:] **DO** int_zmienna_kontrolna = start,stop [, krok]

blok instrukcji

END DO [nazwa]

- nazwa – pomocniczy tekst ułatwiający analizę tekstu programu (coś w rodzaju etykietki)
- int_zmienna_kontrolna – zmienna koniecznie typu Integer
- start, stop - wartość początkowa i końcowa zmiennej_kontrolnej, stała, zmienna, element tablicy etc. lub wyrażenie typu Integer
- krok - przyrost wartości zmiennej_kontrolnej po każdym wykonaniu bloku instrukcji petli, stała, zmienna, element tablicy etc. typu Integer, wartość def.=1.

W praktyce: jako instrukcje pętli rozumie się instrukcję **DO** wraz z instrukcją kończącą pętlę **END DO**.

Przykład:

```
DO i=1,5
    DO j=1,5
      j=i*i
    write (*,*) i,j
  END DO
END DO
```

```
kolumny: DO i=1,n
      wiersze: DO j=1,m
          obraz(i,j)=0
    END DO  wiersze
END DO kolumny
```

Kroki wykonania:

1. wyznaczenie wart. wyrażeń start, stop, krok
2. licznik iteracji = max(int((stop-start+krok)/krok),0)
3. test możliwości wykonania iteracji
   jeżeli licznik iteracji > 0 to zmienna_kontrolna = start
   jeżeli licznik iteracji =0, skok do pierwszej instrukcji czynnej po instrukcji kończącej pętlę
4. wykonanie bloku instrukcji,
5. zmienna_kontrolna = zmienna_kontrolna + krok
6. licznik = licznik -1
7. jeżeli licznik >0 to skok do pkt. 4

Przykład:

```
DO i=1,5
   j=i*i
  write (*,*) i,j
END DO
```

Uwaga: teraz i=6 !!!

PRAWIDŁA

1. **Instrukcja pętli DO (w znaczeniu "praktycznym") może być zawarta w bloku instrukcji wykonywalnych innej instrukcji (np. instrukcji DO).**
2. **Nie wolno zmieniać wartości zmiennej_kontrolnej w bloku instrukcji wykonywanych w petli.**
3. **Każda instrukcja DO (w scisłym sensie) musi mieć „własną” instrukcję END DO kończącą pętle.**
4. **Nie wolno dokonywać skoków do wnętrza "bloku instrukcji" instrukcji DO**

OPTYMALIZACJA PĘTLI

a. Usuwanie wyrażeń niezmienniczych poza pętlę;

b. Łączenie kilku pętli w jedną (tzw. ściskanie pętli);

c. Rozwijanie pętli (multiplikowanie instrukcji w bloku instrukcji);

d. Unikanie pętli;

e. Staranne zagnieżdżanie pętli:

```
      do  k=1,20     !      1 otwarcie
       do  j=1,10    !20 otwarc
        do l=1,5           !200 otwarc
          xx=1             !wykonanie 1000
        enddo        !1000 zamkniec
       enddo         !200 zamkniec
      enddo          !20 zamknięc
!     221 otwarć i 1220 zamknięć
```

```
      do  j=1,10     ! 5 otwarć
       do l=1,20     ! 50 otwarć
          xx=1             !wykonanie 1000
        enddo        !1000 zamknięc
       enddo         !50 zamknięc
      enddo          !5 zamknięc
!     56 otwarć i 1055 zamknięć
```

```
      do  k=1,5            ! 1 otwarcie
```

ETYKIETY

Etykieta instrukcji jest to ciąg cyfr (liczący od 1 do 5 elementów) z których przynajmniej jedna musi być różna od 0.

- Etykiety umieszcza się w dowolnej z kolumn 1-5
- Spacje w etykietach są ignorowane
- W ramach jednego segmentu etykiety instrukcji nie mogą się powtarzać.
- Etykietowane mogą być wszystkie instrukcje.

Poprzez etykiety można się odwoływać tylko do instrukcji wykonywalnych oraz instrukcji FORMAT.

Przykłady: **00001** N1=.... ! OK

1 N1=... ! OK

00000 N1=... **! ERROR**

Instrukcja **READ**

Instrukcja READ przenosi z urządzenia (zbioru zewnętrznego) wartości zmiennych wyspecyfikowanych w jej liście wejścia/wyjścia i przypisuje je odpowiednio elementom listy *imputs*:

READ (io-control-specs) *[inputs]*

inputs – lista zmiennych rozdzielonych , lub instrukcja we/wy typu **do-implikowanego**

Np. **read (*,*) a,b,c,d**
read (*,*) a(23),b,c(12),string1

Instrukcja we/wy typu do-implikowanego ma postać:

(lista_zmiennych , instr_do-implikowanego)

instrukcja do-implikowanego ma postać: kontr = start, end [, krok]

gdzie: kontr, start, end i krok mogą być typu INTEGER lub REAL.

Np. **read (3,'(7f8.2)') a,b,c,(e(i),i=5,11,2)**

READ (*io-control-specs*) *[inputs]*

Lista parametrów opisujących sposób działania instrukcji READ

Opcjonalnie:

a) [UNIT =] numer urządzenia zewnętrznego (zbiór, monitor, klawiatura) – obowiązkowo tylko jeden.
b) [FMT =] specyfikacja formatu (opis sposobu zapisu) **lub** [NML =] nazwa grupy zmiennych
c) REC = numer rekordu do przeczytania (≥1)
d) IOSTAT = zmienna typu integer, 0 gdy brak błędu I/O, numer błędu (>0) gdy błąd I/O, <0 gdy EOF
e) ERR = etyk. instr. czynnej w tym samym segmencie. Wystąpienie błędu I/O lub napotkanie EOF powoduje przekazanie sterowania do tej instrukcji.
f) END = etyk. instr. czynnej w tym samym segmencie. Napotkanie EOF powoduje przekazanie sterowania do tej instrukcji, jeśli nie wystąpił błąd I/O.
g) EOR = etyk. instr. czynnej w tym samym segmencie. Napotkanie EOR lub EOF powoduje przekazanie sterowania do tej instrukcji, jeśli nie wystąpił błąd I/O.

UNIT= można opuścić jeśli jest to pierwszy parametr na liście specyfikacji;
FMT= można opuścić przed specyf. formatu, jeśli jest to drugi parametr na liście specyfikacji;
(NML= można opuścić przed nazwą grupy zmiennych, jeśli jest to drugi parametr na liście specyfikacji)
Jeżeli zastosowany jest specyfikator REC=, nie wolno użyć specyfikatorów END=, NML=, specyfikacją formatu nie może być *.

read (2,fmt='(a6,2f8.3)') str1,r1,r2 **read (3,fmt='(2f10.2)',err=123) r3,r4**

Instrukcja WRITE

Przenosi wartości danych/wyrażeń wyszczególnionych na *liście wejścia/wyjścia* do urządzenia *unit*

WRITE (io-control-specs) iolist

IOLIST - lista nazw danych i wyrażeń, rozdzielonych przecinkami, których wartości mają być przeniesione lub instrukcja we/wy typu do-implikowanego. Lista nie może zawierać struktur ale może zawierać poszczególne elementy struktur.

Np. **write (*,*) a,b,c,d**

Lista parametrów opisujących sposób działania instrukcji WRITE

Opcjonalnie:

1. [UNIT =] numer urządzenia zewnętrznego (zbiór, monitor, klawiatura) – obowiązkowo tylko jeden.
2. [FMT =] specyfikacja formatu (opis sposobu zapisu) **lub** [NML =] nazwa grupy zmiennych
3. REC = numer rekordu do przeczytania (≥1)

4. IOSTAT = zmienna typu integer, 0 gdy brak błędu I/O, numer błędu (>0) gdy błąd I/O, <0 gdy EOF
5. ERR = etyk. instr. czynnej w tym samym segmencie. Wystąpienie błędu I/O lub napotkanie EOF powoduje przekazanie sterowania do tej instrukcji.
6. END = etyk. instr. czynnej w tym samym segmencie. Napotkanie EOF powoduje przekazanie sterowania do tej instrukcji, jeśli nie wystąpił błąd I/O.
7. EOR = etyk. instr. czynnej w tym samym segmencie. Napotkanie EOR lub EOF powoduje przekazanie sterowania do tej instrukcji, jeśli nie wystąpił błąd I/O.

Jeżeli **zapis/odczyt** ma być dokonany do/z **zbioru wewnętrznego** *unitspec* musi być elementem łańcucha znaków, zmienną znakową, tablicą znakową lub jej elementem, znakowym elementem struktury.

np. WRITE (*,1) a,b,c
1 format (3x,3f7.2)

Carriage Control

Pierwszy znak przesyłany w ramach formatowanego rekordu danych do konsoli lub drukarki nie jest drukowany lecz służy do kontroli położenia „głowicy”. Pozostałe znaki są drukowane poczynając od lewego marginesu.

Znak	**Przesunięcie pionowe przed wydrukiem**
0	dwie linie
1	pierwsza linia następnej strony
+	brak
spacja lub inne	jedna linia

PRZYPOMNIENIE RACHUNKU ZDAŃ

Operujemy ***zmiennymi zdaniowymi***, czyli tymi i tylko tymi wyrażeniami, które są prawdziwe lub fałszywe.

funktory rachunku zdań (w malejącej kolejności rozpatrywania):
- negacja, v alternatywa, Λ koniunkcja, → implikacja, ⇔ równoważność

		negacja	alternatywa	koniunkcja	implikacja	równow.
p	**q**	**-p**	**p v q**	**p Λ q**	**p → q**	**p ⇔ q**
0	0	1	0	0	1	1
1	0	0	1	0	0	0
0	1	1	1	0	1	0
1	1	0	1	1	1	1

Przypomnienie: Sprawdzanie wyrażeń logicznych metodą zero-jedynkową:

Sprawdźmy prawo absorpcji: p v (p Λ q) ⇔ p

p	**q**	**p Λ q**	**p v (p Λ q)**	**⇔**
0	**0**	**0**	**0**	**1**
1	**0**	**0**	**1**	**1**
0	**1**	**0**	**0**	**1**
1	**1**	**1**	**1**	**1**

Typ LOGICAL

Zmienna/stała logiczna to obszar pamięci zawierający informację interpretowaną jako reprezentacja wartości logicznej (tzn. prawdy lub fałszu). Jest to jednocześnie wartością tej zmiennej. Zmienna/stała logiczna jest dokładną reprezentacją odpowiadającej jej wartości logicznej.

Słowo kluczowe	Zakres
LOGICAL (def. kind=4)	.TRUE. .FALSE.

LOGICAL [kind=int_exp_init] [,atrybuty] :: nazwa_zmiennej(tablicy)[=wartosc_poczatkowa][,...]
nazwa_funkcji(tablicy)

Stałe logiczne: **.false.** **.true.** **.false._wkind**

```
program foo
implicit none
integer, parameter :: wkind=4
logical :: b, c
logical :: e=.false.
logical (kind=4), parameter :: d=.true.
      b=.false._wkind
      e=d
stop
end program foo
```

OPERACJE LOGICZNE

Wyrażenia logiczne składają się z:

1. danych logicznych (stałych i zmiennych logicznych);

2. operatorów relacji;

3. operatorów logicznych;

Wyrażenia logiczne dają jako wynik wartość logiczną.

Podstawowe rodzaje operandów wyrażeń logicznych:

1. stałe logiczne
2. zmienne logiczne
3. elementy tablic zmiennych logicznych
4. funkcje zwracające wynik typu logicznego
5. wyrażenia relacji
6. stałe i zmienne typu integer
7. elementy struktur typu logicznego

OPERATORY RELACJI

- służą do porównywania wartości dwóch wyrażeń arytmetycznych lub logicznych;
- **wynik porównania jest typu LOGICAL (.TRUE. lub .FALSE.)**
- są operatorami binarnymi.

OPERATOR		ZNACZENIE
<	**.LT.**	mniejsze niż
<=	**.LE.**	mniejsze lub równe
==	**.EQ.**	równe
/=	**.NE.**	nie równe
>	**.GT.**	większe niż
>=	**.GE.**	większe lub równe

Wyrażenia z operatorami relacji dają wyniki typu LOGICAL (.true. lub .false.)

Wszystkie operatory relacji są operatorami dualnymi, umieszczanymi pomiędzy swoimi operandami.

np.: ...(DELTA.LT.0.0)... ...(DELTA<0.0)... ...PIERW1==PIERW2... ...PIERW1.EQ.PIERW2...

Uwagi:

- w skład wyrażeń relacji NIE MOGĄ wchodzić inne wyrażenia relacji, np.:

niech REAL :: a, b, c

wyrażenie **((a .lt. b) . ne. c)** jest błędne, bo wart. logiczna .TRUE./.FALSE./ nie może być mniejsza, większa bądź równa c.

- Jeżeli w wyrażeniu występują liczby różnych typów, to liczba "prostszego typu" przez wyznaczeniem wartości jest przekształcana do typu "bardziej złożonego".
- W wyrażeniach gdzie występują liczby typu COMPLEX, wolno stosować tylko operatory: .EQ. i .NE.
- Wolno porównywać łańcuchy: robi się to metodą porównywania "znak po znaku" ich kodów ASCII.

OPERATORY LOGICZNE

OPERATOR	OPERACJA	RANGA
.NOT.	negacja	1
.AND.	konjunkcja	2
.OR.	alternatywa	3
.XOR.	Alternatywa wyłaczna	4
.EQV.	równoważność	4
.NEQV.	nierównoważność	4

Operatorami wyrażeń logicznych mogą być wyłącznie operatory typu logicznego.

Operacje logiczne o tej samej randze rozpatrywane są w kolejności od lewej do prawej.

operand a i b są:	a .and. b	a .or. b	a .eqv. b	a .xor. b a .neqv. b
oba .true.	.true.	.true.	.true.	.false.
jeden .true. drugi .false.	.false.	.true.	.false.	.true.
oba .false.	.false.	.false.	.true.	.false.

Np.: ((delta=0).and.(lambda<tiny)).or.((delta>0).and.(epsilon==epstiny))

INSTRUKCJE STERUJĄCE „IF”

Instrukcja sterująca IF służy do *uzależnienia* wykonania wybranych fragmentów programu od aktualnej wartości logicznej wyrażenia logicznego będącego jego warunkiem logicznym:

JEŻELI PRAWDZIWY JEST (warunek1) TO

Wykonaj blok_instrukcji_1

A JEŻELI NIE JEST PRAWDZIWY *warunek_1*, A PRAWDZIWY JEST (warunek2) TO

Wykonaj blok_instrukcji_2

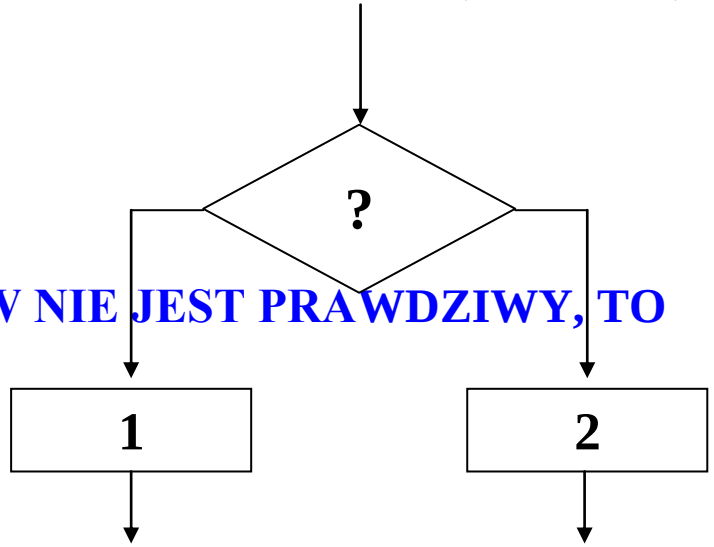
ETC. ETC.

….

A JESLI ŻADEN Z POWYŻSZYCH WARUNKÓW NIE JEST PRAWDZIWY, TO

Wykonaj ten blok instrukcji

KONIEC ROZWAŻAŃ



Instrukcja IF

IF (skalarne_kontrolne_wyrażenie_logiczne) jedna_instrukcja_czynna

Wykonanie instrukcji powoduje:

a) wyznaczenie wartości tzw. kontrolnego wyrażenia logicznego oraz

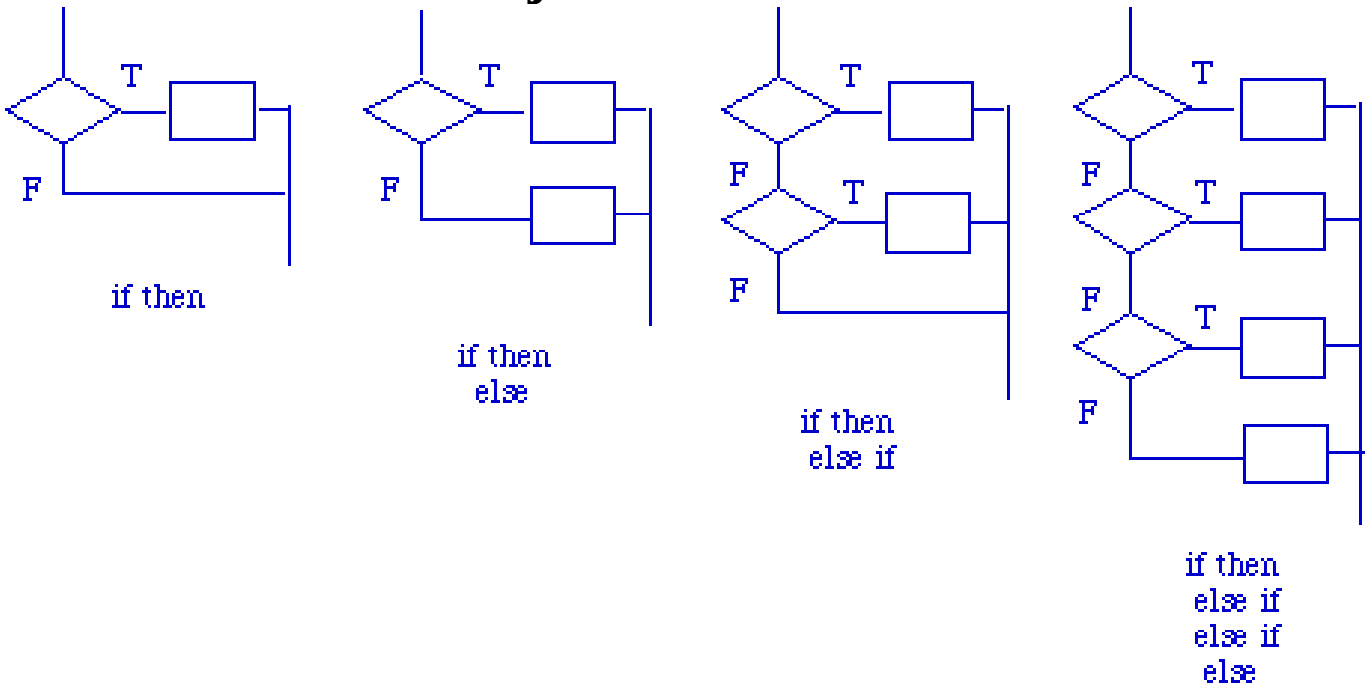
b) wykonanie instrukcji czynnej jeżeli wartością kontrolnego wyrażenia logicznego jest prawda

Instrukcją czynną nie może być instrukcja IF oraz instrukcja END.

Przykłady:

```
Program foo
    implicit none
    real :: k
        read (*,*) k
        if (k<10) k=k**2
        write (*,*) k
    stop
end program foo
```

Instrukcja blokowa IF-THEN



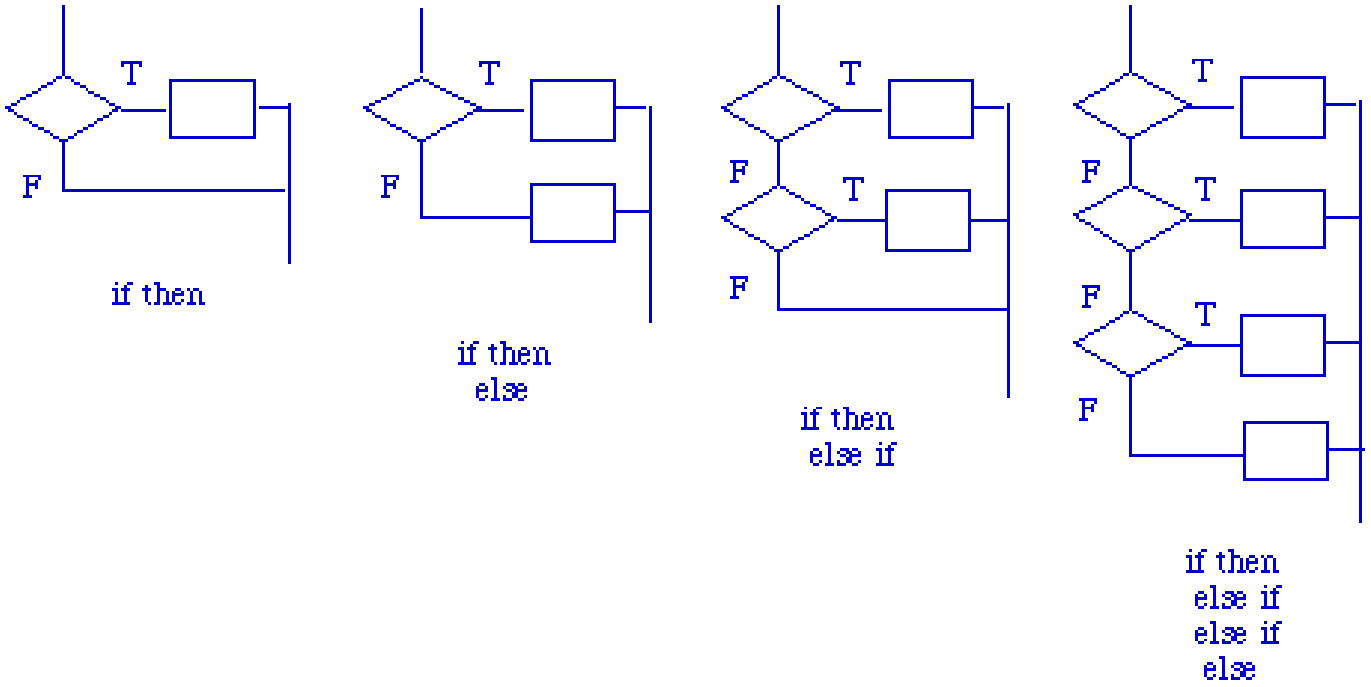
[nazwa:] **IF** (*log_expr*) **THEN**
blok instrukcji
*[***ELSE IF** (*log_expr*) **THEN** *[nazwa]*
blok instrukcji]
...
[**ELSE** *[nazwa]*
blok instrukcji]
END IF *[nazwa]*

Wykonanie instrukcji powoduje:

a) wyznaczenie wartości tzw. kontrolnego wyrażenia logicznego oraz

b) wykonanie instrukcji czynnych bloku if jeżeli wartością kontrolnego wyrażenia logicznego jest prawda

c) przekazanie sterowania do najbliższej instrukcji ELSE / ELSE IF / ENDIF tego samego poziomu.



Wariant uproszczony I (IF-END IF):

Wykonanie instrukcji powoduje wykonanie bloku_instrukcji_IF jeżeli "warunek_logiczny jest prawdziwy", w przeciwnym wypadku sterowanie jest przekazywane do instrukcji END IF.

```
IF (kontrolny_warunek_logiczny ) THEN
    blok_instrukcji
END IF
```

Instrukcja END IF

Instrukcja END IF kończy każdą instrukcje blokową IF, ograniczając zasięg bloku instrukcji wykonywanych warunkowo

- Każda instrukcja blokowa IF musi mieć własną instrukcję kończącą END IF
- Wykonanie instrukcji END IF nie ma innego wpływu na działanie programu.

Przykład:
```
read (*,*) hhmm
if (hhmm<1200) then
    call proc1 ( )
    call proc2 ( )
end if
```

Uwaga: Nie wolno przekazywać kontroli do wnętrza bloku IF

Wariant uproszczony II (IF-ELSE-END IF):

Wykonanie instrukcji IF-ELSE-END IF powoduje:

a) wykonanie "bloku instrukcji IF" jeżeli wartością warunku logicznego jest prawda lub

b) wykonanie "bloku instrukcji ELSE" jeżeli wartością warunku logicznego jest fałsz.

Instrukcja ELSE określa początek bloku_instrukcji_ELSE

.

```
IF (  warunek_logiczny  ) THEN
      blok_instrukcji
ELSE
      blok_instrukcji
END IF
```

Nie wolno przekazywać kontroli do wnętrza bloku ELSE.

Przykład:

```
if (num<0) then
      n1=..
      n2=..
else
      n1=..
      n2=..
endif
```

ZAGNIEŻDŻANIE INSTRUKCJI IF

"Blok instrukcji IF" oraz "blok instrukcji ELSE" mogą zawierać różne warianty instrukcji IF-THEN-ELSE. Instrukcja wewnętrzna (tzw. zagnieżdżona) musi się zawierać całkowicie w jednym z bloków instrukcji zewnętrznej.

```
IF (warunek_1) THEN
    ...
    IF (warunek_2) THEN
        blok_2
    END IF
    ....
ELSE
    ...
    IF (warunek_3) THEN
        blok
    ELSE
        blok
    END IF
    ....
END IF
```

Instrukcja ELSE IF

Jest to wariant instrukcji ELSE dodatkowo uzależniający wykonanie "bloku_instrukcji_ELSE" od spełnienia dodatkowego_warunku_logicznego. Instrukcja ELSE IF umożliwia wykonanie "bloku_instrukcji_ELSE" tylko jeżeli jej warunek_logiczny ma wartość logiczną prawda.

```
ELSE IF (   warunek_logiczny  ) THEN
      blok_instrukcji
ELSE  / ELSE IF / END IF
```

Realizacja przykładowa:

```
IF (warunek_1) THEN
        blok_1_wykonywany_gdy_warunek_1_jest_prawdziwy
    ELSE IF (warunek_2) THEN
        blok_2_wykonywany_gdy_warunek_1_jest_fałszywy_a_warunek_2_prawdziwy
    ELSE
        blok_3_wykonywany_gdy_warunek_1_jest_fałszywy_i_warunek_2_jest_fałszywy
    END IF
```

```
IF (delta .gt. 0) THEN
        x1=-b-sqrt(delta)/(2*a)
        x2=-b+sqrt(delta)/(2*a)
    ELSE IF (delta .eq. 0) THEN
        x1=-b/(2*a)
        x2=x1
    ELSE
        Write (*,*) ‘brak pierw. R’
END IF
```

Typ tablicowy (TABLICE)

Tablica (macierz) jest uporządkowanym zbiorem sąsiadujących i parami rozłącznych zmiennych skalarnych tego samego typu, nazywanych elementami tablicy (tablica jest **rodzajem struktury danych**).

a_{11}	a_{21}	a_{31}	a_{41}	a_{51}	...
a_{12}	a_{22}	a_{32}	a_{42}	a_{52}	...
a_{13}	a_{23}	a_{33}	a_{43}	...	...

W Fortranie elementy tablic składowane są wg. zasady "zapis kolumnami": a(1,1),a(2,1),a(3,1)...

Sposób adresowanie elementów tablicy:

adresy obliczane są względem adresu 1-szego elementu tablicy

a1 a2 a3 a4 a5 a1 a2 a3 a4 a5 .....

konkretny adres względem początku programu

Ilość elementów i ilość wymiarów tablicy zależy od pamięci (**w ELF90 do 7 wymiarów**).

Rodzaje tablic: **STATYCZNE** **i** **DYNAMICZNE**

- ➲ **jawnie deklarowane**
- ➲ **domyślne**
- ➲ **alokowane**
- ➲ **automatyczne**

Tablice muszą być zdefiniowane (tzn. zadeklarowane)
PRZED użyciem

DEKLARACJA TABLICY sposób I

TYP_TABL, DIMENSION ([DOL:] GOR, ...) [ATRYB] :: NAZWA_TABLICY[=WART_POCZ],...

dla każdego wymiaru osobno!

Sam atrybut **DIMENSION**, użyty w ramach instrukcji deklaracji typu, ustala, że:

obiekt jest tablicą statyczną oraz określa ilość i zakres wskaźników.

Przykład deklaracji tablicy: **REAL, DIMENSION (1000) :: TAB1**

Uwaga: zawartość nawiasu nazywamy **deklaratorem rozmiaru** tablicy.

DEKLARACJA TABLICY sposób II

W standardzie F90/95 wolno również użyć deklaracji (**! niedopuszczalne w ELF90 !**):

TYP_TABL [ATRYB] :: NAZWA_TABLICY([DOL:] GOR, ...)

TYP_TABL :: NAZWA_TABLICY([DOL:] GOR, ...) [=WART_POCZ],…

dla każdego wymiaru osobno!

Przykład deklaracji tablicy:

```
REAL :: TAB1(1000) , TAB2(100)
Integer :: a(10:20)
```

Dopuszczane postacie granic dolnych i górnych:

1. stała arytmetyczna;
2. wyrażenie arytmetyczne;
3. (:)

Przykłady deklaracji tablic:

```
REAL, DIMENSION ( 3 , 5 ) :: TAB1                    ! TAB1 jest 2D
INTEGER, DIMENSION (100) :: NUMB, MAMB               ! NUMB i MAMB są 1D
CHARACTER (LEN=10), DIMENSION (1000) :: STARNAMES
REAL, DIMENSION ( 3 :8, 20 ) :: TAB2                 ! TAB2 jest 2D
```

rank (ranga) – ilość wymiarów tablicy (skalary mają rank=0), tab1 ma rank 2

shape (kształt) – rozmiar w każdym z wymiarów, tab1 ma shape (3,5)

size (rozmiar) – ilość elementów tablicy, tab1 ma size 15

```
PROGRAM FOO
IMPLICIT NONE
REAL, DIMENSION (2:5,3:6) :: A
  A(3,4)=1.1
  WRITE(*,*) A(3,4)
STOP
END PROGRAM FOO
```

```
PROGRAM FOO
IMPLICIT NONE
INTEGER, PARAMETER :: I=3, J=6
REAL, DIMENSION (I:J,3:6) :: A
  A(3,4)=1.1
  WRITE(*,*) A(3,4)
STOP
END PROGRAM FOO
```

WSKAZANIE CAŁEJ TABLICY DOKONUJE SIĘ POPRZEZ JEJ NAZWĘ.

WSKAZANIE KONKRETNEGO ELEMENTU / ELEMENTÓW TABLICY:

NAZWA_TABLICY [element_listy_wskaźników **, ...]**

dla każdego wymiaru osobno!

Dopuszczalne postacie elementów listy wskaźników:

Wskaźnik elementu tablicy	⇒	wyrażenie (zmienna) typu INTEGER
Triplet wskaźników	⇒	[wskaźnik_elementu] : [wskaźnik_elementu] : [krok (wyrażenie typu INTEGER)]
Wskaźnik wektorowy	⇒	1-D (*rank*=1), typu INTEGER, wynik operacji na tablicach

➲ Jeżeli każdy ze wskaźników listy elementów tablicy jest wskaźnikiem pojedynczego elementu tablicy, to mamy odwołanie (wskazanie) jednego, konkretnego elementu tablicy.

➲ W przeciwnym wypadku mamy wskazanie wycinka tablicy. Wycinek tablicy ma co najmniej jeden element_listy_wskaźników w postaci tripletu wskaźników lub wskaźnika wektorowego (jednoelementowy wycinek tablicy nie jest skalarem).

Wskazanie pojedynczego elementu tablicy

Każdy ze wskaźników elementu tablicy ⇒ wyrażenie (zmienna) typu INTEGER

```
PROGRAM FOO
IMPLICIT NONE
REAL, DIMENSION (10,10,50) :: TAB1
REAL :: A
INTEGER :: I,J,K
INTEGER, PARAMETER :: L=3
DO I=1,10
  DO J=1,10
    DO K=1,50
      TAB1(I,J,K)=I*J*K
    END DO
  END DO
END DO
A = TAB1 ( 3 , 2*L, L+5 )                    !  A=TAB1(3,6,8)
WRITE (*,*) A
STOP
END PROGRAM FOO
```

Wskazanie wycinka tablicy

Triplet wskaźników ⇒ [wskaźnik_elementu] : [wskaźnik_elementu] : [krok (wyr. typu INTEGER)]

dolna wart. zmian wskaźnika [def. 1]	*górna wart. zmian wskaźnika [def. GOR]*	*przyrost [def. 1]*

a(3:7:2) od 3 do 6 z krokiem 2 (a(3),a(5),a(7))
a(3:6) od 3 do 6 z krokiem 1 (a(3),a(4),a(5),a(6))
a(:5:2) od 1 do 5 z krokiem 2 (a(1),a(3),a(5))
a(:) cała a

```
PROGRAM FOO
IMPLICIT NONE
REAL, DIMENSION (10,10,50) :: TAB1
REAL, DIMENSION(4) :: A
INTEGER ::  I,J,K,L=3
DO I=1,10
  DO J=1,10
    DO K=1,50
      TAB1(I,J,K)=I*J*K
    END DO
  END DO
END DO
A = TAB1 ( 3 , 2*L, 2*L:4*L:2 )        ! 2*L:4*L:2 = od 6 do 12 z krokiem 2
WRITE (*,*) A                                !A(1)=108, A(2)=144, A(3)=180, A(4)=216,   A jest tablica 1-D !!!
STOP
END PROGRAM FOO
```

```
……
REAL, DIMENSION(2,4) :: A
….
A = TAB1 ( 3 , 2*L:2*L+1, 2*L:4*L:2 )          ! A JEST TERAZ 2-D
……
```

Niech A i B będą tablicami:

```
.....
Integer, dimension (2,6) ::  A=13
Integer, dimension (2,2) ::  B
B=a(1:2,3:4:1)
B=a(:,3:4:1)
! jest rownowazne
! b(1,1) = a(1,3)
! b(2,1) = a(2,3)
! b(1,2) = a(1,4)
! b(2,2) = a(2,4)  (=13)
....
```

NADAWANIE WARTOŚCI ELEMENTOM TABLICY

1) Przypisanie

```
Integer, dimension (2,6) ::  A
real :: b=2.3
A(2,4)=3.26*sin(b)
A(I,J)=b
```

2) Podczas deklarowania tablicy:

TYP_TABL, DIMENSION ([DOL:] GOR, ...) [ATRYB] ::
NAZWA_TABLICY[=WART_POCZ],…
TYP_TABL :: NAZWA_TABLICY([DOL:] GOR, ...) [=WART_POCZ],…

a) stała:
```
Integer, dimension (2,6) ::  A=13
integer :: a(10)=2
```

b) wyrażenie arytmetyczne:
```
Real, dimension (6) ::  B=1.+2.*3.
real :: a(20)=2.9*2
```

c) lista wartości, dla tablic jednowymiarowych mająca postać: **(/ lista_wartości /)**

lista wartości składa się z: wyrażeń skalarnych, wyrażeń tablicowych, uproszczonych pętli DO

np. REAL :: A(10)=(/10.,9.,8.,7.,6.,5.,4.,3.,2.,1./)

uproszczona pętla DO:

(lista_wartości, zmienna_kontrol = wyraż_wartpocz, wyraż_wartkonc, [kork])

np.

```
PROGRAM FOO
IMPLICIT NONE
  INTEGER :: I
  REAL, DIMENSION(10) :: A=12.75
  REAL, DIMENSION(20) :: C=(/(10.24, I=1,20)/)
  REAL, DIMENSION(20) :: C=(/(10.24*I, I=1,20)/)
 WRITE (*,*) C(1),C(2)
 WRITE (*,*) D(1),D(2)
STOP
END PROGRAM FOO
```

KONSTRUKCJA WHERE

Konstrukcja **WHERE** umożliwia dokonanie przypisania nowej wartości ***selektywnie*** tylko tym elementom tablicy, dla których tablicowe wyrażenie logiczne ma wartość prawdy.
(tzw. przypisanie selektywne)

WHERE (WYRAZENIE_LOGICZNE_TABLICOWE) TABLICA=WARTOŚĆ

lub

WHERE (wyrazenie_logiczne_tablicowe)
tablica= wyrazenie
[tablica= wyrazenie]...
[*ELSEWHERE*]
tablica= wyrazenie
[tablica= wyrazenie]…
END WHERE

np.:

```
PROGRAM FOO
IMPLICIT NONE
INTEGER :: I
REAL, PARAMETER :: EPSILON=0.05
REAL, DIMENSION(10) :: TB
DO I=1,10
  READ (*,*) TB(I)
END DO

WHERE ( ABS(TB)<EPSILON)
   TB=0
ELSEWHERE
   TB=ABS(TB)
END WHERE
WRITE (*,*) (TB(I),I=1,10)

STOP
END PROGRAM FOO
```

FUNKCJA RESHAPE

Nadawanie wartości początkowych tablicom N-wymiarowym:

RESULT=RESHAPE (SOURCE , SHAPE [, PAD] [, ORDER])

Source – tablica wartości początkowych
Shape – rozmiary i zakres poszczeg. wskaźników
[pad] – wartość uzupełniająca
[order] – permutacja wart. początkowych

Np: REAL, DIMENSION(2,2) :: A = RESHAPE ((/1,2,3,4/), (/2,2/))
source shape

```
PROGRAM FOO
IMPLICIT NONE
REAL, DIMENSION(2,2) :: A=RESHAPE((/1,2,3,4/), (/2,2/))
WRITE (*,*) A(2,2)      ! 4.0
STOP
END PROGRAM FOO
```

WSKAŹNIK WEKTOROWY

Wskaźnik wektorowy to jedno-wymiarowa tablica typu integer, interpretowana jako lista wskaźników elementów innej tablicy

```
PROGRAM FOO
IMPLICIT NONE
REAL :: A(10)=(/10.,9.,8.,7.,6.,5.,4.,3.,2.,1./),B(10)
INTEGER :: V(3)=(/2,4,6/)
INTEGER :: Q(3)=(/1,5,1/)
  WRITE(*,*) A(V)              !  9.0, 7.0, 5.0
  A(V)=3.14
  WRITE (*,*) (A(I),I=1,6))
  B=A(Q)
STOP
END PROGRAM FOO
```

TABLICE DYNAMICZNE

Tablica alokowalna – umieszczana w RAM w miarę potrzeby

Tablica **domyślna** – tablica występująca w podprogramie, której wielkość i rozmiary dopasowywane są do wielkości i rozmiarów tablicy będącej parametrem aktualnym wywołania podprogramu (*).

Tablica **automatyczna** – tablica w podprogramie, której wielkość ustalana jest w zależności od wartości parametru aktualnego wywołania podprogramu (*).

Atrybut **ALLOCATABLE**

Atrybut ALLOCATABLE określa, że wszystkie wymiary tablicy będą określane dynamicznie w trakcie wykonania programu.

Przykład: **REAL, ALLOCATABLE, DIMENSION (: , :) :: TAB1** ! TAB1 jest alokowalną tab. 2D
REAL, ALLOCATABLE :: TAB2 (: , : , :) ! TAB2 jest alokowalną tab. 3D

dla każdego wymiaru osobno!

Atrybut ALLOCATABLE nie może być stosowany do argumentów formalnych podprogramów.

Instrukcja ALLOCATE

Inicjalizuje dynamiczne tablice (tablice **uprzednio zadeklarowane** z atrybutem ALLOCATABLE):

ALLOCATE (nazwa_tablicy ([d:]g,[d:]g,....) [, ...] [,STAT=ier])

d,g - wyrażenia typu integer określające dolne [def.=1] i górne zakresy wskaźników poszczególnych, wcześniej zadeklarowanych wymiarów.
ier - zmienna typu integer zawierająca kod statusu próby inicjalizacji: 0=O.K.

Przykład:

```
PROGRAM FOO
IMPLICIT NONE
INTEGER :: NS
   REAL, ALLOCATABLE, DIMENSION ( : , : ) :: TAB1
    ALLOCATE ( TAB1 (-3:97,100) , STAT=NS )
STOP
END PROGRAM FOO
```

Kolejne re-definicje *d* lub *g* nie wpływają na zakresy wskaźników.

Instrukcja **DEALLOCATE**

Usuwa z pamięci dynamiczne tablice (tablice uprzednio zainicjowane instrukcjami ALLOCATE):

DEALLOCATE (lista_nazw_tablic [,STAT=ier])

lista_nazw_tablic - lista nazw tablic dynamicznych usuwanych z pamięci operacyjnej
ier - zmienna typu integer zawierająca kod statusu próby usunięcia: 0=O.K.

```
C     ALLOCATE (TABLICA(:,..) [,STAT=LAB])
C     WYMIARUJE I UMIESZCZA W PAMIECI TABLICE ALOKOWALNA
C     DEALLOCATE (LISTATABLIC [,STAT=LAB]) ZWALNIA OBSZAR PAMIECI ZAJETY
C           PRZEZ TABLICE ALOKOWALNA
      INTEGER*2 I1
      CHARACTER (LEN=100),  ALLOCATABLE :: OUT(:)
      ...
      I1=25
      ...
      ALLOCATE ( OUT(I1), STAT=II)
      IF (II.NE.0) WRITE (*,*) 'ERROR NR: ',II
      ...
      DEALLOCATE ( OUT, STAT=II)
      IF (II.NE.0) WRITE (*,*) 'ERROR NR: ',II
      ...
      ALLOCATE ( OUT(I1), STAT=II)
      IF (II.NE.0) WRITE (*,*) 'ERROR NR: ',II
      ...
```

FUNKCJA ALLOCATED

Sprawdza, czy tablica alokowalna została umieszczona w RAM-ie

wyn = ALLOCATED (*array*)

wyn – zmienna typu LOGICAL

```
INTEGER, ALLOCATABLE :: I(:,:)
ALLOCATE (I(2,3))
L = ALLOCATED (I)          !     L ← TRUE
```

INSTRUKCJA PĘTLI DO (CZĘŚĆ 2)

[nazwa:] **DO** int_zmienna_kontrolna = start,stop [, krok]

blok instrukcji

END DO [nazwa]

INSTRUKCJA EXIT

Instrukcja EXIT przerywa wykonanie instrukcji pętli DO

EXIT [do-nazwa]

Gdzie**:** *do-nazwa* – nazwa instrukcji Do zawierającej instrukcję EXIT. Jeśli *do-nazwa* jest pominiętą, instrukcja EXIT odnosi się do instrukcji pętli, w której jest bezpośrednio zawarta.

```
outer: do i=1, 10
  inner: do j=1, 10
    if (i>a) exit outer
  end do inner
end do outer
```

```
program foo
      implicit none
      character, parameter :: a*1='f'
      character :: b*1
      integer :: n
      n=0
      do
            read (*,*) b
            if (b==a) exit
            n=n+1
      end do
      write (*,*) n,'  liter przed ‘,a
      stop
end program foo
```

INSTRUKCJA CYCLE

Instrukcja CYCLE powoduje pominięcie bieżącego kroku iteracji w pętli DO

CYCLE [do-nazwa]

Gdzie**:** *do-nazwa* – nazwa instrukcji Do zawierającej instrukcję CYCLE. Jeśli *do-nazwa* jest pominięta, instrukcja CYCLE odnosi się do instrukcji pętli, w której jest bezpośrednio zawarta.

```
program foo
      implicit none
      integer:: n,m
! sumuje liczby ujemne
! dodatnie pomija, zero zatrzymuje
      m=0
      do
        read (*,*) n
        if (n==0) exit
        if (n==abs(n)) cycle
        m=n+m
      end do
      write (*,*) 'suma= ',m
stop
end program foo
```

ZBIORY - FORMATY I SPOSOBY DOSTĘPU

		Format		
		formatowany	**nieformatowany**	**binarny**
Dostęp	**sekwencyjny**	X	X	X
	bezpośredni	X	X	X

Dostęp sekwencyjny:

⇒

rekord nr 1	rekord nr 2	...	rekord nr M	...

Dostęp swobodny:

⇒	rekord nr 1
⇒	rekord nr 2
⇒	...
⇒	...
⇒	rekord nr N

Nb. format zbioru określa się w czasie jego otwierania za pomocą parametru FORM

FORM='formatted' lub **'unformatted'** **np.** open (unit=1,file='foo1', form='formatted')

Dostęp do rekordów zbioru określa się w czasie jego otwierania za pomocą parametru ACCESS

ACCESS='direct' lub **'sequental'**(default) **np. o**pen(unit=1,file='foo2', access='direct')

Jeżeli dostęp jest sekwencyjny to default form='formatted''

Jeżeli dostęp jest bezpośredni to default form='unformated'

INSTRUKCJA OPEN

Podstawowe przeznaczenie: łączy numer jednostki ze zbiorem zewnętrznym lub urządzeniem.

OPEN ([UNIT=] int_wyrażenie [, INNE_PARAMETRY=zmienne] **)**

Np. OPEN (12,file=’foo.dat’)

OPEN (N,FILE=STR1)

Podanie unit=* nie daje żadnego dostrzegalnego efektu ponieważ * jest synonimem klawiatury i ekranu;

Jeżeli unit=unitnew jest taki sam jak unit=unitold, to najpierw zamykany jest unitold a potem otwierany unitnew, czyli otwarcie zbioru z numerem urządzenia wcześniej wykorzystanym powoduje zamknięcie wcześniej otwartego zbioru o tym numerze. Nie można dołączyć zbioru do kilku jednostek jednocześnie.

Łańcuchy znaków, zmienne znakowe, tablice znakowe i ich elementy i inne tablice można traktować jako specjalny rodzaj formatowanych zbiorów (tzw. **zbiory wewnętrzne**). Zbiór wewnętrzny tworzymy zastępując specyfikacje numeru urządzenia nazwą zmiennej znakowej (etc.).

PARAMETR	„WARTOŚĆ”	UWAGI
ACCESS	wyr. znakowe	'Direct', 'sequential' (default)
ERR	etykieta	Etyk. instr. czynnej w tym samym segmencie. Wystąpienie błędu I/O powoduje przekazanie sterowania do tej instrukcji. Przy braku ERR o skutku wystąpienia błędu I/O decyduje IOSTAT.
FILE	wyr. znakowe	Nazwa zbioru.

FORM	wyr. znakowe	'Formatted' lub 'unformatted' dla zbiorów o dost. bezpośrednim def.=’unformatted’ dla zbiorów o dost. sekwencyjnym def.=’formatted’
IOSTAT	zm. integer, elem. tablicy elem. struktury	0 gdy brak błędu I/O; <0 gdy EOF(EOR); numer błędu (>0) gdy błąd I/O
RECL	wyr. integer	długość rekordu;
STATUS	wyr. znakowe	'old', 'new', 'unknown' (default), 'scratch', ‘replace’ • 'old' - zbiór musi istnieć, w przeciwnym przypadku wystąpi błąd I/O; zapis bez przesunięcia się na koniec zbioru powoduje zamazanie jego poprzedniej zawartości; • 'new' - zbiór nie może istnieć, w przeciwnym przypadku wystąpi błąd I/O; • 'scratch' - tymczasowy; • 'replace' - kompilator najpierw próbuje statusu 'old' a potem 'new'.

POSITION	wyr. znakowe	‘rewind’, ‘append’ lub ‘asis’ ‘rewind’ – ustawia wskaźnik na początku otwartego zbioru sekwencyjnego ‘append’ - ustawia wskaźnik przed znakiem EOF ‘asis’ – bez zmian położenia wskaźnika
ACTION	wyr. znakowe	‘read’ ‘write’ ‘readwrite’ (default) ‘read ‘ – zbiór otwarty tylko dla czytania ‘write’ - zbiór otwarty tylko dla zapisu ‘readwrite’ - zbiór otwarty dla czytania i zapisu

```
OPEN (UNIT=1,FILE=’FOO.DAT’,FORM=’FORMATTED’,STATUS=’NEW’,ERR=123,IOSTAT=IST)
OPEN (1,FILE=’FOO.DAT’,ACCESS=’DIRECT’,RECL=25,STATUS=’NEW’,ERR=234,IOSTAT=IST)
NAME=DAYDAT(1:8)//’.DAT’

OPEN (1,FILE=NAME,FORM=’UNFORMATTED’,ACCESS=’DIRECT’,RECL=25,STATUS=’NEW’,
ERR=234, IOSTAT=IST)
```

INSTRUKCJA **REWIND**

Ustawia wskaźnik położenia w zbiorze na początek pierwszego rekordu.

REWIND ([UNIT]=numerzbioru, **ERR**=errlabel, **IOSTAT**=iocheck**])**

numerzbioru	wyrażenie typu INT wyznaczające numer zbioru zewnętrznego
errlabel	etykieta instrukcji czynnej do której przekazywane jest sterowanie w przypadku zaistnienia błędu obsługi zbioru (tzw. błąd I/O)
iocheck	zmienna, elem. tablicy lub struktury zwracający 0 jeżeli instrukcja została wykonana poprawnie i numer błędu, jeżeli wystąpił błąd I/O

```
PROGRAM FOO
IMPLICIT NONE
CHARACTER (LEN=1) :: A,B,C,D,E
OPEN
(1,FILE='FOO.DAT',STATUS='SCRATCH')
WRITE (1,'(F6.2)') 12.34
REWIND (1)
WRITE (1,'(F6.2)') 98.76
REWIND(1)
READ (1,'(6A1)') E,A,B,E,C,D
WRITE (*,*) D,C,E,B,A
CLOSE(1)
STOP
END PROGRAM FOO
```

INSTRUKCJA **CLOSE**

Instrukcja CLOSE usuwa przypisanie zbioru do numerem urządzenia.

CLOSE ([**unit=**]numerunit [**,ERR=**errorlab][**,IOSTAT=**iocheck][**,STATUS=**stat])

ERR	etykieta	Etyk. instr. czynnej w tym samym segmencie. Wystąpienie błędu I/O powoduje przekazanie sterowania do tej instrukcji. Przy braku ERR o skutku wystąpienia błędu I/O decyduje IOSTAT.
IOSTAT	zm. integer, elem. tablicy, struktury	0 gdy brak błędu I/O, numer błędu (>0) gdy błąd I/O
STATUS	wyr. znakowe	'keep', 'delete' • 'keep' - zamyka i pozostawia **(default)**; • 'delete' - zamyka i usuwa; • 'scratch' - tymczasowy - zawsze 'delete'

close (7)

close(8,status='delete')

close (9,status='delete',iostat=ns)

INSTRUKCJA **INQUIRE**

Podaje informacje o cechach zbioru zewnętrznego lub urządzenia.

INQUIRE ([**UNIT=**]int_unitspec lub **FILE=**filename, **INNE_PARAMETRY=** ZMIENNA **)**

INNE_PARAMETRY (wybrane): ZMIENNA

ACCESS	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'sequental' dla zb. sekwenc. 'direct' dla dost. bezpośredn. ‘undefined’ jeśli zbiór nie jest przypisany do numeru urządzenia
BINARY	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' dla zb. binarnego 'no' lub 'unknown' w innym wypadku
OPPENED	zm. logiczna, elem. tablicy lub struktury, zwraca: .true. jesli zb. jest otwarty dla operacji I/O, .false. w przeciwnym wypadku
NAMED	zm. logiczna, elem. tablicy lub struktury, zwraca: .true. jesli zb. ma nazwę, .false. w przeciwnym wypadku
DIRECT	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jesli możliwy jest dost. bezpośredni 'no' lub 'unknown' w innym wypadku
ERR	etykieta instrukcji czynnej do której przekazywane jest sterowanie w wypadku błędu I/O.

EXIST	zm. logiczna, elem. tablicy lub struktury, zwraca: .true. jesli zb. istnieje .false. jeśli brak zbioru
FORM	zm. znakowa lub elem. tablicy; zwraca: 'formatted' jeżeli zbiór przyłączony jest do formatowanej operacji I/O 'unformatted' jeżeli zbiór przyłączony jest do nieformatowanej operacji I/O 'binary' jeżeli zbiór przyłączony jest do binarnej operacji I/O'
IOSTAT	zm. całkowita, elem. tablicy lub struktury, zwraca 0 jeśli nie wystąpił błąd, numer błędu w przypadku wystąpienia błędu I/O
NAME	zmienna znakowa, elem. tablicy lub struktury; zwraca nazwę zbioru
NEXTREC	zm. całkowita, elem. tablicy lub struktury, zwraca numer kolejnego rekordu w zbiorach o dost. Bezpośrednim
NUMBER	zm. całkowita, elem. tablicy lub struktury, zwraca numer urządzenia/zbioru
RECL	zm. całkowita, elem. tablicy lub struktury, zwraca dług. rekordu zbioru o dost. bezpośrednim
SEQUENTIAL	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jesli możliwy jest dost. sekwencyjny 'no' lub 'unknown' w innym wypadku
UNFORMATTED	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jeśli możliwy jest dostęp nieformat. 'no' jeśli nie jest możliwy dostęp nieformat. lub 'unknown' w innym wypadku

FORMATTED	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jeśli możliwy jest dostęp format. 'no' jeśli nie jest możliwy dostęp format. lub 'unknown' w innym wypadku
READ	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jeśli możliwy jest odczyt 'no' jeśli nie jest możliwy odczyt lub 'unknown' w innym wypadku
WRITE	zmienna znakowa, elem. tablicy lub struktury; zwraca: 'yes' jeśli możliwy jest zapis 'no' jeśli nie jest możliwy zapis lub 'unknown' w innym wypadku
READWRITE	

```
inquire (unit=8, access=acc, err=200)   ! access met. dla unit 8? goto 200 gdy error
inquire (tunit, opened=opnd, direct=dir) ! is tunit otwarty? Dozwol. direct access?
inquire (file="foo.dat", recl=record_length) ! reclen w zbiorze "foo.dat"?
```

```
PROGRAM FOO
IMPLICIT NONE
CHARACTER (LEN=12) :: NAME
LOGICAL :: YESNO
YESNO=.FALSE.
    DO WHILE (.NOT. YESNO)
        WRITE (*,*) 'PODAJ NAZWĘ ZBIORU:'
        READ (*,'(A)') NAME
        INQUIRE (FILE=NAME, EXIST=YESNO)
        IF (.NOT. YESNO) WRITE (*,*) 'NIE MA TAKIEGO ZB.'
    END DO
```

	STOP **END PROGRAM FOO**

FORMATOWANE WEJŚCIE / WYJŚCIE

Explicite opis formatu danych zformatowanych wyprowadzanych do zbioru lub wprowadzanych ze zbioru podawany jest za pomocą list_deskryptorów_formatu czyli list specyfikacji sposób zapisu/odczytu danych.

SPOSOBY SPECYFIKACJI FORMATU

1. Instrukcja FORMAT (etykieta jako specyfikator formatu)

WRITE (*,label) lista_i/o

labelFORMAT (lista_deskryptorów_formatu)

uwagi: instrukcja FORMAT musi być etykietowana, błędy w liście_deskryptorów formatu powodują błędy ujawiniające się podczas kompilacji lub wykonania

2. Wyrażenie znakowe lub stała znakowa jako specyfikator formatu

WRITE (*,'(wyrażenie_znakowe_tworzące_listę_deskryptorów_formatu)')

WRITE (*,'(stałą_znakowa_będąca_listą_deskryptorów_formatu)')

3. * jako specyfikator formatu

WRITE (*,*) lista_i/o

Instrukcja **FORMAT**

Ustala format (sposób zapisu/odczytu) danych zapisywanych do zbioru lub ze zbioru czytanych.

etykieta FORMAT ([lista_deskryptorów_formatu])

WRITE (*,fmt='listy_deskryptorów_formatu'**) lista_obiektów**
READ (*,fmt='lista_deskryptorów_formatu'**) lista_obiektów**

WRITE (*,lab**) lista_obiektów**
READ (*,lab**) lista_obiektów**
Lab FORMAT (lista_deskryptorów_formatu**)**

Lista_deskryptorów_formatu jest stałą znakową. Ograniczona jest dwoma ' (albo dwoma ") jeśli występuje w instrukcjach READ i WRITE. Składa się z rozdzielonych przecinkami deskryptorów formatów. Lista nie może być pusta. Instrukcja FORMAT musi być etykietowana.

Deskryptory formatu: n-krotnie powtarzalne

jednokrotne

n-krotnie powturzona lista_deskryptorów_formatu (w nawiasach, dopuszczalne jest do 16 poziomów zagłębienia nawiasów).

Np: nDESKR = DESKR, DESKR, DESKR, ... itd. n-razy

n(DESKR1,mDESKR2) = DESKR1, mDESKR2,DESKR1, mDESKR2,DESKR1, mDESKR2, ... n-razy

DESKRYPTORY JEDNOKROTNE

Postać	**Działanie**	**Wejście**	**Wyjście**
' *łańcuch znaków* '	Przesyła *łańcuch znaków* do wyjścia	Nie	Tak
:	Przerywa interpretację formatu jeśli na liście wejścia-wyjścia nie ma następnych elementów	Nie	Tak

Uwagi:

1. W *łańcuchu znaków:* spacje są znaczące i są wyprowadzane, podwójny apostrof w *łańcuchu znaków* jest uzywany dla reprezentacji pojedynczego *apostrofu wyprowadzanego

```
write (*,*) 'Ala ma "kota"'
write (*,999)
```

```
999  format (' Ala ma "kota"')
     write (*,'(" Ala ma ""kota""")')
```

2. Znak / oznacza koniec transferu danych do/z bierzącego rekordu. Podczas czytania: wskaźnik położenia w zbiorze przesuwany jest na początek następnego rekordu. Podczas pisania: wpisywany jest znak końca rekordu i

wskaźnik położenia w zbiorze przesuwany jest na początek następnego rekordu.

```
     write (*,999)
999  format(1x,'k',/,1x,'o',/,' l',/' u',/,1x,'m',/,1x,'n',/,1x,'a')
```

3. Znak : przerywa interpretację formatu jeśli na liście wejścia-wyjścia nie ma nastepnych elementów.

```
      program foo
      implicit none
      real :: a=2.0, b=3.0, c=4.0 ,d=5.0
            write (*,777) a,b,c,d
            write (*,777) a,b
777               format (' a=',f5.2,' b=',f5.2,' c=',f5.2,' d=',f5.2)
            write (*,555) a,b,c,d
            write (*,555) a,b
555               format (' a=',f5.2:'b=',f5.2:'c=',f5.2:'d=',f5.2)
      stop
      end program foo
```

POWTARZALNE DESKRYPTORY FORMATU

Iw[.m]	liczby całkowite
Zw, Ow, Bw	liczby hexa, oktalne, binarne
Fw.d	liczby rzeczywiste
Ew.d[Ee]	liczby rzeczywiste z wykładnikiem
EN*w.d[*E*e]*	liczby rzeczywiste z wykładnikiem, zapis inżynierski
Es*w.d[*E*e]*	liczby rzeczywiste z wykładnikiem, zapis naukowy
Gw.d[Ee]	ogólnie liczby (w tym całkowite)
Dw.d	liczby rzeczywiste podwójnej precyzji
Lw	watrości logiczne
A[w]	łańcuchy
n/	koniec transferu danych do/z bierzącego rekordu

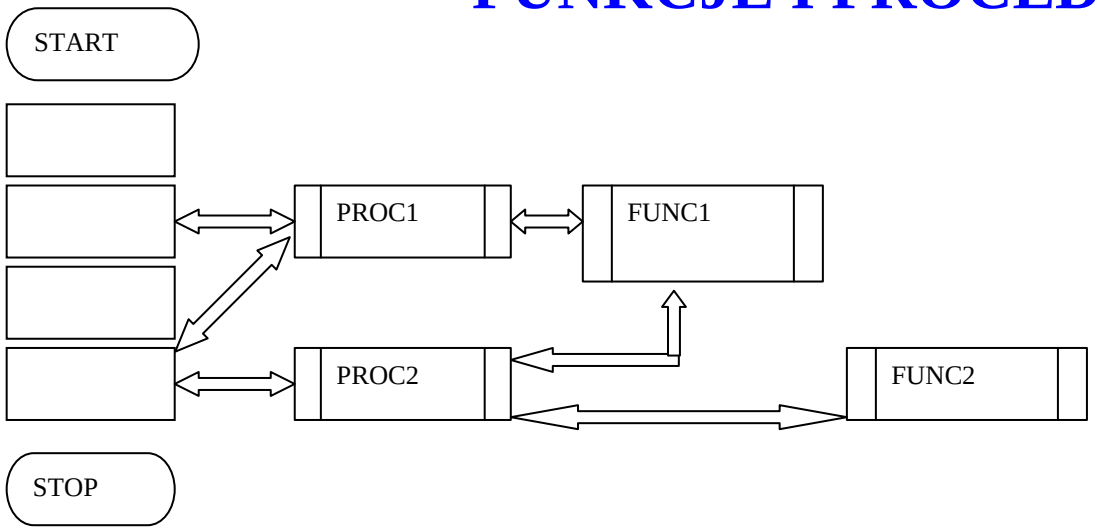
Powtarzalne deskryptory formatu służą do specyfikacji formatu wprowadzania/wyprowadzania danych ww. typów.

Podstawowe zasady:

1. Podczas czytania: pola zawierające tylko spacje interpretowane są jako zero.
2. Znak + jest opcjonalny.
3. Podczas czytania wg. deskryptorów **F, E, G, D** wystąpienie *explicite* kropki dziesiętnej w polu zawierającym zapis liczby znosi wszelkie specyfikacje położenia kropki dziesiętnej podane w deskryptorze.
4. Jeśli ilość znaków wyprowadzonych jest większa niż szerokość pola, pole wypełniane jest *.
5. Podczas wyprowadzania liczb rzeczywistych część ułamkowa jest tak zaokrąglana, by jej zapis po zaokrągleniu miał długość zgodną z wyspecyfikowaną w deskryptorze.
6. Dla sformatowania jednej liczby zespolonej niezbędne są dwa kolejne deskryptory (**F, E, G, D**)

Deskryptory formatu mogą być zawierane w nawiasach. Nawias poprzedziny stała typu integer oznacza powtórzenie analizy listy desktyptorów odpowiednia liczbę razy.

FUNKCJE I PROCEDURY



Fortran umożliwia wydzielanie określonych wydzielonych segmentów kodu w postaci podprogramów (segmentów).
Procedury mogą mieć postać **funkcji** (ang. *function*) i **procedur** (ang. *subroutine*):

1. **zaimplementowanych** (ang. *intrinsic*) – gotowe do użycia, dostarczone z kompilatorem
2. **wewnętrznych** – zawarte w pewnym segmencie kodu i wywoływane tylko z tego segmentu.
3. **zewnętrznych** – stanowiące niezależne segmenty kodu, niezależnie kompilowane i wywoływane z dowolnego segmentu programu.

   Podprogramy wewnętrzne i zewnętrzne mogą być **rekursywne**.

   Podprogramy wewnętrzne i zewnętrzne mogą być ***specyficzne*** (ang. *specific*) lub ***ogólne*** (ang. *generic*).

PROCEDURY I FUNKCJE ZAIMPLEMENTOWANE

LISTA PONAD 100 "ZAIMPLEMENTOWANYCH" FUNKCJI I PROCEDUR

Procedury i funkcje zaimplementowane mogą mieć jako argument skalar lub tablicę. W przypadku tablicy wszystkie elementy tablicy są przetwarzane tak samo, jak skalar.

Table 26: Numeric Functions

Name	Function Type	Argument Type	Description	Class
ABS	Numeric	Numeric	Absolute Value.	Elemental
AIMAG	REAL	COMPLEX	Imaginary part of a complex number.	Elemental
AINT	REAL	REAL	Truncation to a whole number.	Elemental
ANINT	REAL	REAL	REAL representation of the nearest whole number.	Elemental
CEILING	INTEGER_4	REAL	Smallest INTEGER greater than or equal to a number.	Elemental
CMPLX	COMPLEX	Numeric	Convert to type COMPLEX.	Elemental
CONJG	COMPLEX	COMPLEX	Conjugate of a complex number.	Elemental
DIM	INTEGER or REAL	INTEGER or REAL	The difference between two numbers if the difference is positive; zero otherwise.	Elemental

Table 27: Mathematical Functions

Name	Function Type	Argument Type	Description	Class
ACOS	REAL	REAL	Arccosine.	Elemental
ASIN	REAL	REAL	Arcsine.	Elemental
ATAN	REAL	REAL	Arctangent.	Elemental
ATAN2	REAL	REAL	Arctangent of y/x (principal value of the argument of the complex number (x,y)).	Elemental
COS	REAL or COMPLEX	REAL or COMPLEX	Cosine.	Elemental
COSH	REAL	REAL	Hyperbolic cosine.	Elemental
EXP	REAL or COMPLEX	REAL or COMPLEX	Exponential.	Elemental
LOG	REAL or COMPLEX	REAL or COMPLEX	Natural logarithm.	Elemental
LOG10	REAL	REAL	Common logarithm.	Elemental
SIN	REAL or COMPLEX	REAL or COMPLEX	Sine.	Elemental

Table 28: Character Functions

Name	Description	Class
ACHAR	Character in a specified position of the ASCII collating sequence.	Elemental
ADJUSTL	Adjust to the left, removing leading blanks and inserting trailing blanks.	Elemental
ADJUSTR	Adjust to the right, removing trailing blanks and inserting leading blanks.	Elemental
CHAR	Given character in the collating sequence of the a given character set.	Elemental
IACHAR	Position of a character in the ASCII collating sequence.	Elemental
ICHAR	Position of a character in the processor collating sequence associated with the kind of the character.	Elemental
INDEX	Starting position of a substring within a string.	Elemental
LEN	Length of a CHARACTER data object.	Inquiry
LEN_TRIM	Length of a CHARACTER entity without trailing blanks.	Elemental
LGE	Test whether a string is lexically greater than or equal to another string based on the ASCII collating sequence.	Elemental

Table 29: Array Functions

Name	Description	Class
ALL	Determine whether all values in a mask are true along a given dimension.	Transformational
ALLOCATED	Indicate whether an allocatable array has been allocated.	Inquiry
ANY	Determine whether any values are true in a mask along a given dimension.	Transformational
COUNT	Count the number of true elements in a mask along a given dimension.	Transformational
CSHIFT	Circular shift of all rank one sections in an array. Elements shifted out at one end are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.	Transformational
DOT_ PRODUCT	Dot-product multiplication of vectors.	Transformational
EOSHIFT	End-off shift of all rank one sections in an array. Elements are shifted out at one end and copies of boundary values are shifted in at the other. Different sections can be shifted by different amounts and in different directions by using an array-valued shift.	Transformational

Table 30: Inquiry and Kind Functions

Name	Description	Class
ALLOCATED	Indicate whether an allocatable array has been allocated.	Inquiry
ASSOCIATED	Indicate whether a pointer is associated with a target.	Inquiry
BIT_SIZE	Size, in bits, of a data object of type INTEGER.	Inquiry
DIGITS	Number of significant binary digits.	Inquiry
EPSILON	Positive value that is almost negligible compared to unity.	Inquiry
HUGE	Largest representable number of data type.	Inquiry
KIND	Kind type parameter.	Inquiry
LBOUND	Lower bounds of an array or a dimension of an array.	Inquiry
LEN	Length of a CHARACTER data object.	Inquiry
MAXEXPONENT	Maximum binary exponent of data type.	Inquiry
MINEXPONENT	Minimum binary exponent of data type.	Inquiry
PRECISION	Decimal precision of data type.	Inquiry

Table 31: Bit Manipulation Procedures

Name	Function Type	Argument Type	Description	Class
BTEST	LOGICAL_4	INTEGER_4	Bit testing.	Elemental
IAND	INTEGER	INTEGER	Bit-wise logical AND.	Elemental
IBCLR	INTEGER	INTEGER	Clear one bit to zero.	Elemental
IBITS	INTEGER	INTEGER	Extract a sequence of bits.	Elemental
IBSET	INTEGER	INTEGER	Set a bit to one.	Elemental
IEOR	INTEGER	INTEGER	Bit-wise logical exclusive OR.	Elemental
IOR	INTEGER	INTEGER	Bit-wise logical inclusive OR.	Elemental
ISHFT	INTEGER	INTEGER	Bit-wise shift.	Elemental
ISHFTC	INTEGER	INTEGER	Bit-wise circular shift of rightmost bits.	Elemental
MVBITS		INTEGER	Copy a sequence of bits from one INTEGER data object to another.	Subroutine

Table 33: Standard Intrinsic Subroutines

Name	Description	Class
DATE_AND_ TIME	Date and real-time clock data.	Subroutine
MVBITS	Copy a sequence of bits from one INTEGER data object to another.	Subroutine
RANDOM_ NUMBER	Uniformly distributed pseudorandom number or numbers in the range $0 \le x < 1$.	Subroutine
RANDOM_ SEED	Set or query the pseudorandom number generator used by RANDOM_NUMBER. If no argument is present, the processor sets the seed to a predetermined value.	Subroutine
SYSTEM_ CLOCK	INTEGER data from the real-time clock.	Subroutine

FUNKCJA WEWNĘTRZNA

Funkcja wewnętrzna – wydzielony zestaw instrukcji służący do (najczęściej wielokrotnego) wykonania powtarzalnego procesu obliczeniowego w ramach wybranego segmentu kodu. Kod funkcji wewnętrznej programista zawiera w tym segmencie kodu i przez to funkcja wewnętrzna może być wywoływana tylko z tego segmentu.

Po wywołaniu funkcji wewnętrznej:

- sterowanie przekazywane jest do jej nagłówka;
- następnie wykonywane są jej instrukcje czynne (w kolejności zgodnej z ogólnymi zasadami);
- wewnątrz segmentu funkcji nazwa_funkcji zachowuje się jak zmienna i należy dokonać co najmniej raz **explicite** przypisania jej jakiejś wartości (wynik obliczeń);
- wartość przypisana nazwie funkcji jest zwracana **(pojedyncza wartość skalarna lub pojedyncza tablica)** w miejscu wywołania funkcji;
- po wykonaniu wszystkich koniecznych instrukcji czynnych sterowanie przekazywane jest do pierwszej instrukcji czynnej po instrukcji, która wywołała funkcję.

Składniki segmentu funkcji wewnętrznej:

instrukcja **FUNCTION name**

deklaracje argumentów formalnych

deklaracja typu funkcji

instrukcje wykonywalne (w tym nazwa_funkcji=... oraz RETURN)

instrukcja **END FUNCTION name**

```
program foo
implicit none
real :: r1,r2
read (*,*) r1,r2
write (*,*) ad(r1,r2)
stop
      CONTAINS
      function ad (r1, r2)
            implicit none
            real :: ad
            real, intent(in) :: r1, r2
            ad=r1+r2
            return
      end function ad
end program foo
```

INSTRUKCJA FUNCTION

Instrukcja FUNCTION identyfikuje dany fragment kodu jako funkcję oraz służy do zadeklarowania nazwy tej funkcji, argumentów formalnych oraz ew. rekursywności funkcji.

[RECURSIVE] FUNCTION *name* (*lista arg. formalnych*) [RESULT (resname)]

np.: **function suma_pierw (a,b)**

INSTRUKCJA RETURN

Instrukcja **RETURN** kończy wykonanie funkcji (lub procedury) i przekazuje sterowanie z powrotem do instrukcji następującej po instrukcji wywołującej funkcję.

RETURN

ELF90 wymaga poprzedzenia instrukcji końca podprogramu instrukcją RETURN. W podprogramie można użyć w razie potrzeby wielokrotnie instrukcji RETURN

INSTRUKCJA END FUNCTION

Instrukcja END FUNCTION identyfikuje koniec kodu funkcji.

END FUNCTION name

W FORTRANIE argumenty przekazywane są do podprogramów przez referencję. Dla uniknięcia niepożądanych efektów ubocznych argumenty formalne **muszą być** podczas deklarowania opatrzone odpowiednim atrybutem INTENT:

Atrybut INTENT

INTENT(IN) – argument jest przeznaczony do przekazania danych do funkcji i nie może przekazywać danych z funkcji do segmentu funkcje wywołującego.

INTEND(OUT) - argument jest przeznaczony do przekazania danych do segmentu funkcje wywołującego i nie może przekazywac danych do funkcji (*).

INTENT(IN OUT) - argument jest przeznaczony do przekazania danych w obu kierunkach (*).

(*) – odnosi się do procedur.

Np.:

```
function ad (r1, r2)
    implicit none
    real :: ad
    real, intent(in) :: r1, r2
    ad=r1+r2
    return
end function ad
```

Argumenty ze słowem kluczowym (KEYWORDS ARGUMENTS)

Pozwala wyspecyfikować powiązanie pomiędzy parametrami aktualnymi wywołania a parametrami formalnymi funkcji tak, że bez znaczenia jest kolejność na liście.

([.....,] ARG_FORMALNY = ARG_AKTUALNY [,....])

```
program foo
implicit none
write (*,*) rozn (r2=3.0,r1=5.0)
stop
```

```
    CONTAINS
    function rozn (r1, r2)
        implicit none
        real :: rozn
        real, intent(in) :: r1, r2
        rozn=r1-r2
        return
    end function rozn
end program foo
```

Argumenty OPCJONALNE

Argumenty opcjonalne mogą, lecz nie musza być dostarczane funkcji w momencie jej wywołania.

Opatruje się takie

zmienne przy deklaracji atrybutem OPTIONAL.

```
program foo
implicit none
write (*,*) rozn2 (r2=3.0,r1=5.0, r3=1.0)    ! write (*,*) rozn2 (r2=3.0, r1=5.0)    !write (*,*) rozn2(10.0, 5.0)
stop
      CONTAINS
      function rozn2 (r1, r2, r3)
            implicit none
            real :: rozn2
            real, intent(in) :: r1, r2
            real, intent(in), optional :: r3
            if (present(r3)) then
              rozn2=r1-r2-r3
            else
              rozn2=r1-r2
            end if
            return
      end function rozn2
end program foo
```

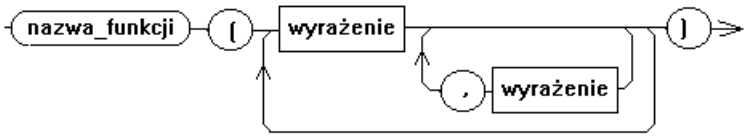
INSTRUKCJA CONTAINS

Instrukcja CONTAINS oddziela kod programu, modułu lub procedury od kodu zawartych w nich procedur (instrukcja ta jest instrukcją bierną).

CONTAINS

```
program foo
implicit none
real :: r1,r2
read (*,*) r1,r2
write (*,*) ad(r1,r2)
stop
CONTAINS
function ad (r1, r2)
      implicit none
      real :: ad
      real, intent(in) :: r1, r2
      ad=r1+r2
      return
end function ad
end program foo
```

WYWOŁANIE FUNKCJI WEWNĘTRZNEJ



<table>
<tr><td colspan="2">PROGRAM, FUNCTION, SUBROUTINE, or MODULE statement</td></tr>
<tr><td colspan="2">USE statements</td></tr>
<tr><td rowspan="4">FORMAT
statements</td><td>IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td></tr>
<tr><td>Derived-type definitions,
interface blocks,
type declaration statements,
and specification statements</td></tr>
<tr><td>Executable statements</td></tr>
<tr><td colspan="2">CONTAINS statement</td></tr>
<tr><td colspan="2">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="2">END program unit statement</td></tr>
</table>

FUNKCJE ZEWNĘTRZNE

Funkcja zewnętrzna jest to segment programu, który może być wywołany z innego segmentu.

Po wywołaniu funkcji zewnętrznej:

- sterowanie przekazywane jest do jej nagłówka;
- następnie wykonywane są jej instrukcje czynne (w kolejności zgodnej z ogólnymi zasadami);
- wewnątrz segmentu funkcji nazwa funkcji zachowuje się jak zmienna i należy dokonać **co najmniej raz** explicite przypisania jej jakiejś wartości;
- wynik działania funkcji jest potem zwracany **(pojedyncza wartość skalarna lub pojedyncza tablica)** w miejsce wywołania funkcji;
- po wykonaniu wszystkich koniecznych instrukcji czynnych funkcji sterowanie przekazywane jest do pierwszej instrukcji czynnej po instrukcji, która wywołała procedurę.

TWORZENIE SEGMENTU FUNKCJI ZEWNĘTRZNEJ

Funkcje zewnętrzne (słowo kluczowe **FUNCTION**)

Słowo kluczowe FUNCTION identyfikuje dany segment programu jako funkcję oraz służy do zadeklarowania nazwy tego segmentu, argumentów formalnych oraz ew. rekursywności funkcji.

[RECURSIVE] FUNCTION *name* (*lista arg. formalnych*) [RESULT (resname)]

np.: **function kl (a,b)**

Składniki segmentu funkcji zewnętrznej (external function):

instrucja **FUNCTION name ..**

instrukcje **USE**

deklaracje argumentów formalnych

instrukcje wykonywalne (w tym obowiązkowo nazwa_funkcji=...)

def. funkcji wewnętrznych

instrukcja **END FUNCTION name**

Z zasady: kolejność, ilość i typy argumentów w wywołaniu funkcji muszą się zgadzać z kolejnością, liczbą i typami odpowiednich arg. formalnych.

Np.: kompletna funkcja zewnętrzna dodająca dwie liczby

```
function ad (r1, r2)
  implicit none
  real :: ad
  real, intent(in) :: r1, r2
  ad=r1+r2
  return
end function ad
```

Funkcje zewnętrzne mogą być wywoływane z dowolnego miejsca w dowolnym segmencie programu, ale:

W ELF90 wykorzystanie funkcji (i procedury) zewnętrznej jest możliwe tylko po umieszczeniu tzw.

„interface'u" w segmencie wywołującym.

[Najlepsza metoda programowania to umieszczenie „interface" w module
i użycie tego modułu w każdym z segmentów wywołujących tą procedurę.]

"Interface" zawiera wszystkie informacje o cechach procedury, niezbędne dla kompilatora dla poprawnego jej wywołania:

1. nazwa procedury
2. liczba, kolejność, typ, przeznaczenie parametrów,
3. ew. opcjonalność parametrów lub informacje, że są wskaźnikami (pointer'ami)
4. jeśli jest to funkcja, to również opis typu wyniku.

<table>
<tr><td colspan="2">PROGRAM, FUNCTION, SUBROUTINE, or MODULE statement</td></tr>
<tr><td colspan="2">USE statements</td></tr>
<tr><td rowspan="4">FORMAT
statements</td><td>IMPLICIT NONE</td></tr>
<tr><td>PARAMETER
statements</td></tr>
<tr><td>Derived-type definitions,
interface blocks,
type declaration statements,
and specification statements</td></tr>
<tr><td>Executable statements</td></tr>
<tr><td colspan="2">CONTAINS statement</td></tr>
<tr><td colspan="2">Internal subprograms or module subprograms</td></tr>
<tr><td colspan="2">END program unit statement</td></tr>
</table>

Blok interface'u specyfikuje sposób wywołania procedury. Może być również użyty do zdefiniowania (przeciążenia) operatora lub zdefiniowania (przeciążenia) podstawienia.

Składnia bloku interface'u:

instrukcja INTERFACE

[treść interface'u: nagłówek procedury
deklaracje argumentów formalnych,
zakończenie procedury]

instrukcja END INTERFACE

Jako "podręczną" regułę można przyjąć, iż "interface" to po prostu procedura z wyciętymi deklaracjami jej zmiennych lokalnych i wszystkimi instrukcjami algorytmu.

```
program foo
implicit none
interface
  function ad (r1, r2)
    implicit none
    real :: ad
    real, intent(in) :: r1, r2
  end function ad
  function sub (r1, r2)
    implicit none
    real :: sub
    real, intent(in) :: r1, r2
  end function sub
end interface
real :: r1,r2
read (*,*) r1,r2
write (*,*) ad(r1,r2)
write (*,*) sub(r1,r2)
stop
end program foo
```

WYWOŁANIE FUNKCJI ZEWNĘTRZNEJ

```
program foo
implicit none
interface
  function ad (r1, r2)
    implicit none
    real :: ad
    real, intent(in) :: r1, r2
  end function ad
  function sub (r1, r2)
    implicit none
    real :: sub
    real, intent(in) :: r1, r2
  end function sub
end interface
real :: r1,r2
read (*,*) r1,r2
write (*,*) ad(r1,r2)
write (*,*) sub(r1,r2)
stop
end program foo
```

```
function ad (r1, r2)
  implicit none
  real :: ad
  real, intent(in) :: r1, r2
  ad=r1+r2
  return
end function ad

function sub (r1, r2)
  implicit none
  real :: sub
  real, intent(in) :: r1, r2
  sub=r1-r2
  return
end function sub
```

INSTRUKCJA SUBROUTINE

Instrukcja SUBROUTINE identyfikuje dany segment programu jako procedurę oraz służy do zadeklarowania nazwy tego segmentu, argumentów formalnych oraz ew. rekursywności procedury.

[RECURSIVE] SUBROUTINE *podprogram-name* (*[lista_arg-formal]*) [RESULT(resname)]

Podprogram-name - nazwa podprogramu, globalna w całym segmencie. Nie może być użyta dla oznaczenie innego obiektu.

lista arg. formalnych - lista argumentów - lista formalnych argumentów podprogramu. Argumenty rozdziela się przecinkami.

Typy argumentów deklaruje się w zwykły sposób **wewnątrz** segmentu funkcji

Resmane – nazwa zmiennej zwracającej wynik funkcji rekursywnej.

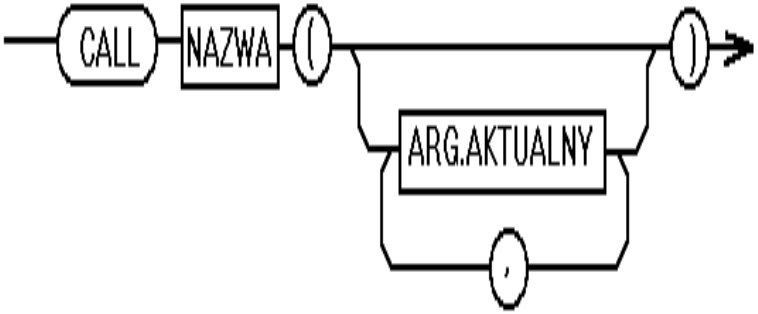
```
subroutine ad (r1, r2, r3)
implicit none
real, intent(in) :: r1, r2
real, intent(out):: r3
r3=r1+r2
return
end subroutine ad
```

INSTRUKCJA CALL

Instrukcja CALL służy do wywołania podprogramu

Działanie instrukcji CALL:

1. Wyznaczana jest wartość argumentów podprogramu będących wyrażeniami
2. Argumenty aktualne SA łączone z argumentami formalnymi
3. Wykonywany jest podprogram
4. Sterowanie jest przekazywane do następnej instrukcji czynnej w segmencie wywołującym



Atrybut INTENT

INTENT(IN) – argument jest przeznaczony do przekazania danych do podprogramu i nie może przekazywać danych z podprogramu do segmentu wywołującego.

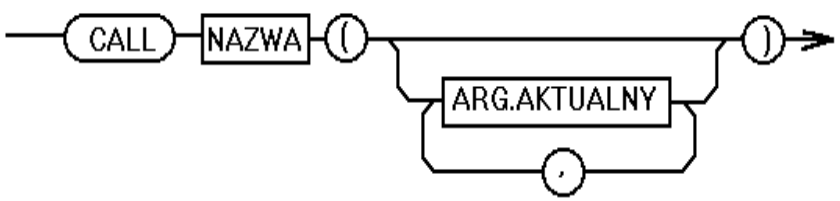
INTEND(OUT) - argument jest przeznaczony do przekazania danych do segmentu wywołującego i nie może przekazywać danych do podprogramu.

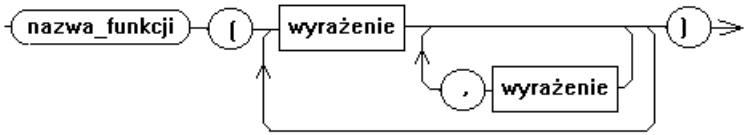
INTENT(IN OUT) - argument jest przeznaczony do przekazania danych w obu kierunkach.

Np.:

```
subroutine ad (r1, r2, r3)
implicit none
real, intent(in) :: r1, r2
real, intent(out):: r3
r3=r1+r2
return
end subroutine ad
```

WYWOŁYWANIE PROCEDUR I FUNKCJI





Co może znajdować się na liście argumentów aktualnych?

- **stała, zmienna, wyrażenie**
- **skalar, tablica, podprogram, wskaźnik**

Obowiązuje <u>zgodność typów</u> argumentu formalnego i odpowiadajacego mu argumentu aktualnego.

TABLICE jako argumenty aktualne wywołania

Argumentem formalnym w podprogramie może być:

- **tablicą o *explicite* zadeklarowanej strukturze, np. foo(5:9,2:21) lub foo(n,m,k) (automatyczną)**
- **tablicą o *przybranej* formie, np. foo(0:,5:), foo(:,:,:)**
- **tablicą o *przybranym* rozmiarze, np. foo(2,4:10,*)**

1. Tablica o explicite zadeklarowanej strukturze

```
SUBROUTINE P1 (TAB,SUM)
 IMPLICIT NONE
 REAL, INTENT(IN) :: TAB(100)
 REAL, INTENT(OUT) :: SUM
 INTEGER :: I
  SUM=0.0
  DO I=1,100
   SUM=SUM+TAB(I)
 END DO
 RETURN
END SUBROUTINE P1
```

```
PROGRAM FOO1
  IMPLICIT NONE
  REAL :: MAT_A(5,5,4)=4.0
  REAL :: MAT_B(100,100)=20.0, TOTAL
   INTERFACE
     SUBROUTINE P1 (TAB,SUM)
     IMPLICIT NONE
     REAL, INTENT(IN) :: TAB(100)
     REAL, INTENT(OUT) :: SUM
     END SUBROUTINE P1
   END INTERFACE
  CALL P1 (MAT_A,TOTAL)  ! MAT_A ma ten sam rozmiar ale inną
  WRITE (*,*) TOTAL      !  ilość wymiarów niż TAB
  CALL P1 (MAT_B,TOTAL)  ! MAT_B ma inny rozmiar i inną ilość
  WRITE (*,*) TOTAL      ! wymiarow niż TAB
  STOP
END PROGRAM FOO1
```

Zawsze argument aktualny musi mieć rozmiar ≥ rozmiaru argumentu formmalnego!

Granice zmienności indeksów tablicy będącej argumentem formalnym podprogramu mogą być także argumentami formalnymi tegoż podprogramu. Taką tablicą nazywamy automatyczną (rodzaj tablicy dynamicznej).

```
PROGRAM FOO2
  IMPLICIT NONE
  REAL :: MAT_A(5,5,4)=5.0, MAT_B(100,100)=10.0,
TOTAL
  INTEGER :: N
  N=100
  CALL P1 (MAT_A,TOTAL,N)     ! TAB stanie się tablicą
  WRITE (*,*) TOTAL           ! 100-elementową
  CALL P1 (MAT_B,TOTAL,10000)  ! TAB stanie sie
  WRITE (*,*) TOTAL            ! tablica 10000-elem.
  STOP
   CONTAINS
      SUBROUTINE P1 (TAB,SUM,N)
       IMPLICIT NONE
       INTEGER, INTENT(IN) :: N
       INTEGER :: I
       REAL, INTENT(IN) :: TAB(N)
       REAL, INTENT(OUT) :: SUM
        SUM=0.0
        DO I=1,N
          SUM=SUM+TAB(I)
        END DO
       RETURN
      END SUBROUTINE P1
END PROGRAM FOO2
```

```
PROGRAM FOO2A
     IMPLICIT NONE
     REAL :: MAT(10,10,10)=5.0, TOTAL
     INTEGER :: N=5,M=3,L=6,K=2
     CALL P1 (MAT,N,M,L,K,TOTAL)
     WRITE (*,*) TOTAL
     STOP
      CONTAINS
        SUBROUTINE P1 (TAB,N,M,L,K,SUM)
         IMPLICIT NONE
         INTEGER, INTENT(IN) :: N,M,L,K
         INTEGER :: I,J,F
         REAL, INTENT(IN) :: TAB(N,M:L,K:L)
         REAL, INTENT(OUT) :: SUM
          SUM=0.0
          DO I=1,N
            DO J=M,L
              DO F=K,L
               SUM=SUM+TAB(I,J,F)
              END DO
            END DO
          END DO
         RETURN
        END SUBROUTINE P1
   END PROGRAM FOO2A
```

2. Tablicą o *przybranej* formie

Znamy wymiar tablicy ale nie znamy rozmiaru → można wykorzystać tablicę o *przybranej* formie

REAL, DIMENSION(0:,5:,:,:) :: A – tablica 4-ro wymiarowa, dolne zakresy wsk. pierwszego i drugiego są określone jako 0 i 5.

INTEGER, DIMENSION(:,:) :: B – tablica 2-wymiarowa

Brak określenia dolnej granicy wskaźnika oznacza z def. = 1

TABLICA O PRZYBRANEJ FORMIE BĘDĄCA ARGUMENTEM AKTUALNYM MUSI MIEĆ TEN SAM WYMIAR ALE ZAKRES INDEKSÓW JEST DOWOLNY

```
PROGRAM FOO3
  IMPLICIT NONE
  REAL :: A(5,10)=5.0, B(5,10)
  CALL P2 (A, B)
  WRITE (*,*) B(1,1)
  STOP
  CONTAINS
    SUBROUTINE P2 (M1, M2)
    IMPLICIT NONE
    REAL, INTENT(IN) :: M1(:,:)
    REAL, INTENT(OUT) :: M2(:,:)
       M2=M1*M1
    RETURN
    END SUBROUTINE P2
END PROGRAM FOO3
```

```
PROGRAM FOO4
 IMPLICIT NONE
 REAL :: A(50,100)=5.0, B(5,10)
 CALL P2 (A(16:20,51:60), B)   ! wymiar musi być ten sam, zakres wskaźników
 WRITE (*,*) B(1,1)            ! jest dowolny
 STOP
 CONTAINS
   SUBROUTINE P2 (M1, M2)
   IMPLICIT NONE
   REAL, INTENT(IN) :: M1(:,:)
   REAL, INTENT(OUT) :: M2(:,:)
      M2=M1
   RETURN
   END SUBROUTINE P2
END PROGRAM FOO4
```

1. Tablicą o *przybranym rozmiarze*

Znamy wymiar tablicy i wszystkie zakresy zmiennosci wskaźników oprócz ostatniego → można wykorzystać tablicę o *przybranym* rozmiarze

```
PROGRAM FOO5
    IMPLICIT NONE
    INTEGER (KIND=4) :: N
    REAL :: A(10,100)=5.0, B(10,10)=4.0, SUM
        N=1000
        CALL P2 (A, N, SUM)
        WRITE (*,*) SUM
        CALL P2 (B, 100, SUM)
        WRITE (*,*) SUM
        STOP
            CONTAINS
            SUBROUTINE P2 (M, N, TOTAL)
            IMPLICIT NONE
            INTEGER (KIND=4), INTENT(IN) :: N
            INTEGER (KIND=4) :: I
            REAL, INTENT(IN) :: M(*)
            REAL, INTENT(OUT) :: TOTAL
                TOTAL=0.0
                DO I=1,N
                    TOTAL=TOTAL+M(I)
                END DO
                RETURN
            END SUBROUTINE P2
END PROGRAM FOO5
```

na marginesie:

```
PROGRAM FOO5A
    IMPLICIT NONE
    REAL :: A(10,100)=5.0, B(10,10)=4.0
    WRITE (*,*) SUM(A)
    WRITE (*,*) SUM(B)
    STOP
END PROGRAM FOO5A
```

robi to samo, ale szybciej się go pisze ☺

Tablice o przybranym rozmiarze mają dokładnie określone granice zmienności wszystkich indeksów (a więc i wymiar) oprócz górnej granicy ostatniego indeksu:

```
INTEGER :: A(2:6,3:5,*)
INTEGER :: D(6,2:5,5:*)
INTEGER :: D(N,M:L,K:*)
```

„ŻYWOTNOŚĆ” ZMIENNYCH LOKALNYCH W PODPROGRAMACH

Zmienne lokalne w podprogramach zachowują swoja wartośc tylko w czasie działania tych podprogramów, chyba, że użycie atrybutu SAVE spowoduje przechowanie wartości zmiennej lokalnej pomiedzy kolejnymi wywołaniami podprogramu.

```
PROGRAM FOO6
IMPLICIT NONE
  CALL S()
  CALL S()
  CALL S()
  STOP
  CONTAINS
    SUBROUTINE S()
      IMPLICIT NONE
      INTEGER :: N
      N=N+1
      WRITE (*,*) N
      RETURN
    END SUBROUTINE S
END PROGRAM FOO6
```

rezultat: 4981234, 4981234, 4981234

```
PROGRAM FOO6A
IMPLICIT NONE
  CALL S()
  CALL S()
  CALL S()
  STOP
  CONTAINS
    SUBROUTINE S()
      IMPLICIT NONE
      INTEGER, SAVE :: N
      N=N+1
      WRITE (*,*) N
      RETURN
    END SUBROUTINE S
END PROGRAM FOO6A
```

rezultat: 1, 2, 3

Zadanie: wytłumaczyć rezultat otrzymany w lewej kolumnie.

```
PROGRAM FOO7
IMPLICIT NONE
character :: a*20
  write (*,1)
  read (*,*) a
  call s(a)
  write (*,1)
  read (*,*) a
  call s(a)
  write (*,1)
  read (*,*) a
  call s(a)
1 format (' Podaj klienta: ')
  stop
  contains
    subroutine s(a)
      implicit none
      character, intent(in) :: a*20
      integer, save :: sum
      if ((a(1:5) == 'Kowal') .or. (a(1:5) == 'kowal')) then
        sum=sum+1
        write (*,2) sum
      end if
2     format (' Znaleziono:',i3,' Kowal(a/i)',/)
      return
    end subroutine s
end program foo7
```

```
Podaj klienta: Glowacz

Podaj klienta: Kowal

Znaleziono:  1 Kowal(a/i)

Podaj klienta: Kowalczyk

Znaleziono:  2 Kowal(a/i)

Program completed
Press Enter to Continue.
```

Dlaczego Kowalczyk został zaliczony do Kowali?

PODPROGRAMY REKURSYWNE

Podprogram rekursywny to podprogram który może wywołać „sam siebie”.
Podprogramy wywołujące same siebie muszą być deklarowane z użyciem słowa kluczowego **RECURSIVE**. Podprogramy wywołujące same siebie *explicite* muszą być deklarowane z podaniem zmiennej pomocniczej **RESULT**.

FUNKCJA REKURSYWNA

Instrukcja RECURSIVE FUNCTION identyfikuje dany segment programu jako funkcję rekursywną oraz służy do zadeklarowania nazwy tego segmentu, argumentów formalnych.

RECURSIVE FUNCTION *name* (*lista arg. formalnych*) RESULT (resname)

PROCEDURA REKURSYWNA

Instrukcja RECURSIVE SUBROUTINE identyfikuje dany segment programu jako procedurę rekursywną oraz służy do zadeklarowania nazwy tego segmentu, argumentów formalnych.

RECURSIVE SUBROUTINE *podprogram-name* (*[lista_arg-formal]*) RESULT(resname)

Należy użyć słów kluczowych:
RECURSIVE – deklaracja podprogramu rekursywnego
RESULT (zmienna) – deklaracja nazwy wyniku funkcji
Np.: RECURSIVE FUNCTION P1 (A, B) RESULT (RECRES)

n! dla liczb naturalnych definiuje się *rekurencyjnie* jako:

1 gdy n=0

n! = n*(n-1)! gdy n>0

n=6 ➔ 6! = 6*5! = 6*5*4! = 6*5*4*3! = 6*5*4*3*2! = 6*5*4*3*2*1! = 6*5*4*3*2*1*0! = 720.

```
PROGRAM FOO8
IMPLICIT NONE
WRITE (*,*) SILNIA(6)
STOP
      CONTAINS
      RECURSIVE FUNCTION SILNIA (N) RESULT
(NS)
      IMPLICIT NONE
      INTEGER :: NS
! uwaga na deklaracje zmiennych, brak deklaracji typu
funkcji
      INTEGER, INTENT(IN) :: N
      IF (N < 0) THEN
        RETURN
      ELSE IF (N == 0) THEN
        NS=1
        RETURN
      ELSE
        NS=N*SILNIA(N-1)
      END IF
      RETURN
      END FUNCTION SILNIA
END PROGRAM FOO8
```

```
PROGRAM FOO9
IMPLICIT NONE
WRITE (*,*) FACTORIAL (3)
STOP
      CONTAINS
      RECURSIVE FUNCTION FACTORIAL(I)
RESULT(FACT)
            IMPLICIT NONE
            INTEGER :: FACT
            INTEGER, INTENT(IN) :: I
            IF (I == 1) THEN
                  FACT = 1
                  RETURN
            END IF
            FACT=I*FACTORIAL(I-1)
            RETURN
      END FUNCTION FACTORIAL
END PROGRAM FOO9
```

Uwaga: procedury rekursywne bywają bardzo nieefektywne czasowo!

MODUŁY

```
MODULE STALMATFIZ
  IMPLICIT NONE
  REAL :: PI=3.14159
  REAL :: G=9.81
END MODULE STALMATFIZ
```

MODUŁ

MODUŁ jest to segment programu, którego „mechanizmy” mogą być wykorzystywane przez inne segmenty programu, jeśli tylko moduł został do nich „przyłączony”.

Moduł może zawierać:

1. **Deklaracje zmiennych globalnych** (w ramach programu, - obiekt zadeklarowany w module jest widoczny w każdym z segmentów programów, do którego przyłączono moduł):
2. **Deklaracje zmiennych nie będących zmiennymi globalnymi**;
3. **Interfejsy procedur**;
4. **Deklaracje procedur**: w module mogą być zawarte procedury, widoczne dla wszystkich segmentów, do którego przyłączono moduł. Procedury te są explicite podane w module i dlatego nie trzeba dla nich umieszczać interface’ów.
5. **„Pakiety” narzędzi:** typy pochodne zmiennych, operatory pochodne, procedury i interface mogą być zdefiniowane i zawarte w module, co pozwala używać ich w sposób „obiektowy”
6. **Rozszerzenia semantyki języka**: zestawy zdefiniowanych w module przez użytkownika typów pochodnych, metod dostępu i przeciążonych operatorów i funkcji wrodzonych pozwala używać ich jak integralnych części języka.

Procedury zaimplementowane w modułach - tak jak funkcje wewnętrzne - mają z definicji *explicite* interfejsy czyli blok interfejsu *nie jest* dla nich potrzebny.

Moduł nie może zawierać instrukcji programu.
Moduł nie może istnieć samodzielnie, musi być używany przez program lub inny moduł.

Składnia modułu:

```
MODULE nazwa_modułu
    instrukcje public/private
    instrukcje deklaracji i specyfikacje
  [CONTAINS
    definicje procedur modułu]
END MODULE nazwa_modułu
```

INSTRUKCJA MODULE

Instrukcja MODULE rozpoczyna segment programu typu moduł.

MODULE nazwa_modułu

Nazwa modułu musi być unikatowa: różna od nazw innych segmentów programu, procedur zewnętrznych i nazw lokalnych w module.

```
MODULE STALMATFIZ
  IMPLICIT NONE
  REAL :: PI=3.14159
  REAL :: G=9.81
END MODULE STALMATFIZ
```

Moduły mogą być wykorzystywane dzięki użyciu instrukcji USE, umieszczonej jako pierwsza spośród instrukcji specyfikacji.

Czyli: moduł/podprogram/program używa (instrukcja USE) pewien moduł, który też może używać (instr. USE) inny moduł itd.

INSTRUKCJA USE

Instrukcja USE wskazuje, iż moduł przez nią wskazany jest dostępny dla segmentu, w którym została użyta. Umożliwia także ewentualną ***zmianę nazwy*** oraz ***ograniczenie dostępności*** elementów modułu.

USE nazwa-modułu [**,** wykorzystywana_lokalna_nazwa_elementu_modułu **=>** nazwa_elementu_modułu]

lub

USE nazwa_modułu **, ONLY:** [nazwa_elem_modułu lub OPERATOR lub ASSIGNMENT (=)]

Instrukcja USE z parametrem ONLY udostępnia tylko te elementy modułu, które występują na „liście" parametru. Jeżeli w segmencie występuje kilka instrukcji USE, listy zmian nazw oraz listy parametru ONLY są łączone.

```
use moj_mod, aleph => alpha   ! alpha jest nazywana aleph aby zapobiec konfliktowi z alpha w innym module!
use twoj_mod, only: alpha, beta, operator(+) ! uzyj z tego modulu tylko alpha, beta i zdefiniowany operator(+)
```

```
MODULE MODUL1
IMPLICIT NONE

CONTAINS

  FUNCTION AD (R1, R2)
    IMPLICIT NONE
    REAL :: AD
    REAL, INTENT(IN) :: R1, R2
    AD=R1+R2
    RETURN
  END FUNCTION AD

  FUNCTION SUB (R1, R2)
    IMPLICIT NONE
    REAL :: SUB
    REAL, INTENT(IN) :: R1, R2
    SUB=R1-R2
    RETURN
  END FUNCTION SUB

END MODULE MODUL1
```

```
PROGRAM FOO9
USE MODUL1
IMPLICIT NONE
REAL :: X1=5, X2=6, WYN
  WYN=AD(X1,X2)-SUB(X2,X1)
  WRITE (*,*) WYN
STOP
END PROGRAM FOO9
```

```
MODULE MODUL1
IMPLICIT NONE
CONTAINS
 FUNCTION AD (R1, R2)
  IMPLICIT NONE
  REAL :: AD
  REAL, INTENT(IN) :: R1, R2
  AD=R1+R2
  RETURN
 END FUNCTION AD
 FUNCTION SUB (R1, R2)
  IMPLICIT NONE
  REAL :: SUB
  REAL, INTENT(IN) :: R1, R2
  SUB=R1-R2
  RETURN
 END FUNCTION SUB
END MODULE MODUL1
```

```
SUBROUTINE FOOSUB (X,Y)
      USE MODUL1
      IMPLICIT NONE
      REAL, INTENT(IN) :: X
      REAL, INTENT(OUT) :: Y
      Y=SUB(X,2*X)+AD(X,2*X)
      RETURN
END SUBROUTINE FOOSUB

PROGRAM FOO10
      USE MODUL1
      IMPLICIT NONE
      REAL :: Y

      INTERFACE
       SUBROUTINE FOOSUB (X,Y)   ! podprogram foosub nie jest zawarty
        USE MODUL1               ! w module, wymaga explicite interfac'u
            IMPLICIT NONE
            REAL, INTENT(IN) :: X
            REAL, INTENT(OUT) :: Y
       END SUBROUTINE FOOSUB
      END INTERFACE

         WYN=AD(X1,X2)-SUB(X2,X1)
         WRITE (*,*) WYN
         CALL FOOSUB (WYN, Y)
         WRITE (*,*) Y
         STOP
END PROGRAM FOO10
```

INSTRUKCJA PRIVATE

Instrukcja **PRIVATE** specyfikuje obiekty dostępne tylko wewnątrz danego modułu.
Może być użyta tylko w modułach.

PRIVATE [:: lista_nazw]

Lista_nazw jest listą nazw wcześniej zadeklarowanych w segmencie, procedur lub OPERATOR(operator) lub ASSIGNMENT(=)

```
MODULE FOO1
      IMPLICIT NONE
      REAL :: A, B, C
      PRIVATE :: A              ! A JEST DOSTEPNE TYLKO WEWNĄTRZ MODUŁU
END MODULE FOO1
```

INSTRUKCJA PUBLIC

Instrukcja **PUBLIC** specyfikuje obiekty dostępne z segmentu programu używającego dany moduł.
Może być użyta tylko w modułach.

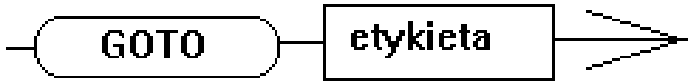
PUBLIC [:: lista_nazw]

```
MODULE FOO2
      IMPLICIT NONE
      PRIVATE                   ! TERAZ DEFAULT DOSTĘP JEST „PRIVATE”
      REAL :: A, B, C
      PUBLIC :: A         ! A JEST DOSTEPNA NA ZEWNĄTRZ MODUŁU
END MODULE FOO2
```

WYKOPALISKA

INSTRUKCJA GOTO SKOKU "BEZWARUNKOWEGO"

Wskazuje wybraną etykietowaną instrukcję czynną jako następną do wykonania (przekazuje sterowanie programem do wskazanej instrukcji)



- Etykieta musi opatrywać instrukcję zawartą w tym samym segmencie programu co instrukcja GOTO;
- Nie wolno skakać do wnętrza instrukcji „DO” oraz instrukcji warunkowych „IF”.

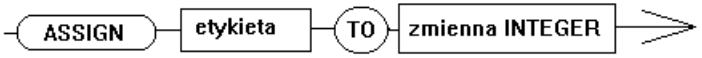
Elegancki przykład, jak NIE należy pisać programów:

```
PROGRAM FOO7
IMPLICIT NONE
INTEGER :: N
4 WRITE (*,7) ' PODAJ LICZBE: '
READ (*,*) N
IF (N<0) THEN
  GO TO 1
ELSE IF (N==0) THEN
  GO TO 2
ELSE
  GO TO 3
END IF
1 WRITE (*,6) ' MNIEJ NIZ ZERO'
GO TO 4
2 WRITE (*,6) ' ZERO'
GO TO 5
3 WRITE (*,6) ' PONAD ZERO'
GO TO 4
5 WRITE (*,6) ' TRAFIONY'
6 FORMAT (A15,/)
7 FORMAT (A15)
STOP
END PROGRAM FOO7
```

Co jeszcze można spotkać w dziedzinie GOTO? (czyli wykopaliska)

Instrukcja ASSIGN

Instrukcja ASSIGN przypisuje zmiennej typu całkowitego etykietę



Uwaga: etykieta nie jest typu integer, oznacza wyłącznie samą siebie!!!

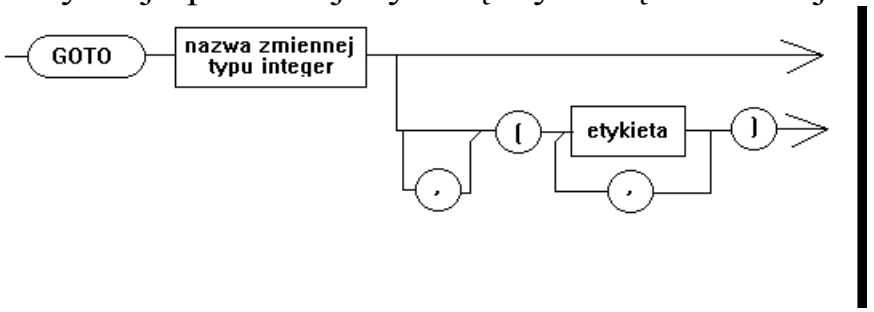
- Etykieta musi opatrywać instrukcję czynną, zawartą w tym samym segmencie programu co instrukcja ASSIGN;
- Tak określona zmienną typu INTEGER można rozumieć jako rónoważna etykiecie (czyli w praktyce jest ona etykietą (adresem) instrukcji).

np:

```
      INTEGER :: n1
      ASSIGN 400 TO n1
400   write (*,*) 'n1 wcale nie równa sie 400!!!'
```

Instrukcja GOTO skoku "przypisanego"

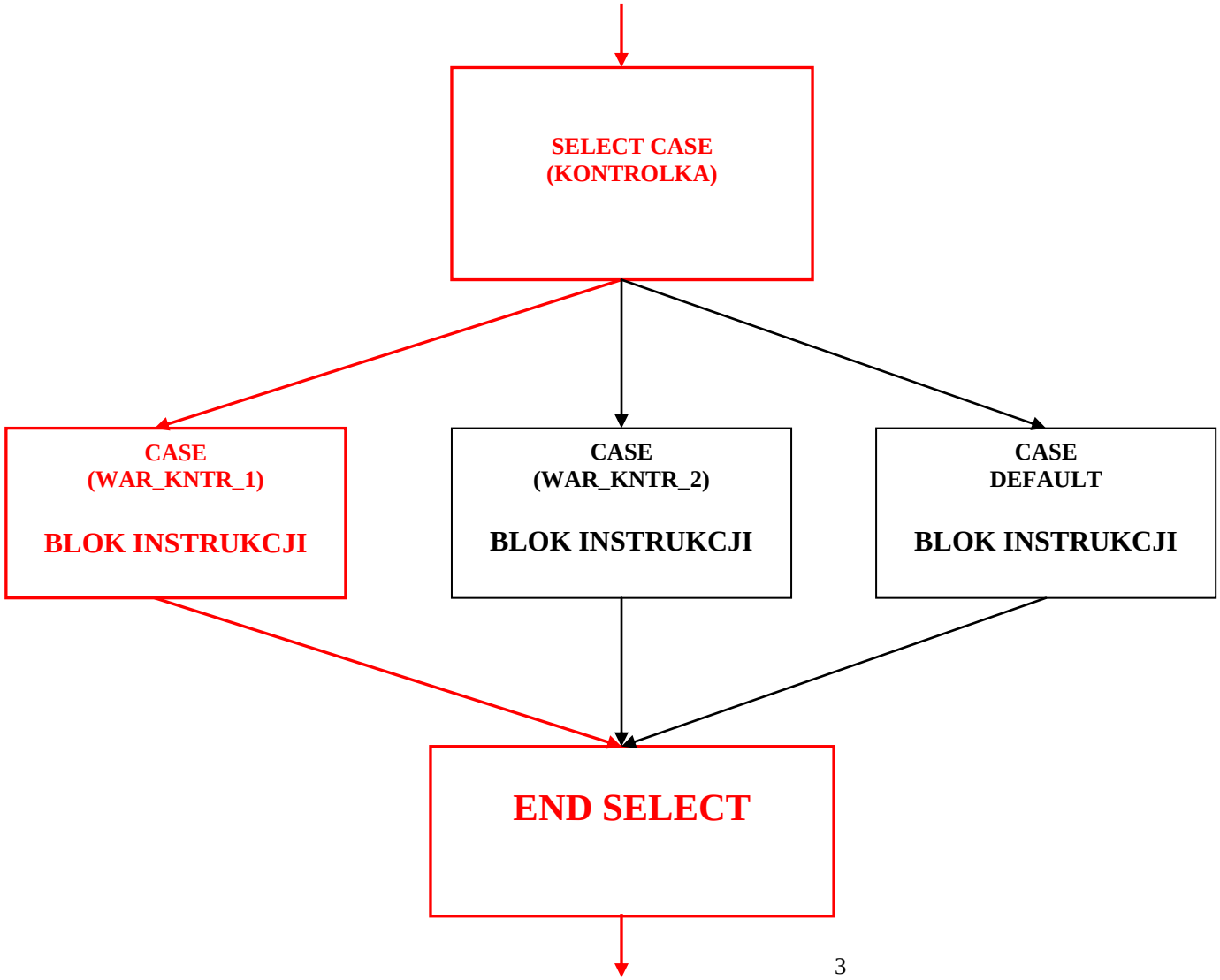
Przekazuje sterowanie do instrukcji czynnej opatrzonej etykietą wybraną wcześniej za pomocą instrukcji ASSIGN.



- Etykieta musi opatrywać instrukcję zawartą w tym samym segmencie programu co instrukcja GOTO;
- Nie wolno skakać do wnętrza instrukcji DO oraz instrukcji warunkowych "IF".

INSTRUKCJA CASE

Instrukcja CASE służy do wyboru jednego z *bloków instrukcji wykonywalnych* na podstawie aktualnej wartości *wyrażenia kontrolnego*.



Składnia:

```
[ nazwa : ] SELECT CASE (wyrażenie-kontrolne)
                CASE (wartość-kontrolna [,wartość-kontrolna]  ... ) [ nazwa ]
                    blok instrukcji
                    ...
                [CASE DEFAULT [ nazwa ]]
                    block
                END SELECT [ nazwa ]
```

nazwa opcjonalna nazwa instrukcji CASE.

wyrażenie-kontrolne wyrażenie skalarne typu INTEGER lub CHARACTER.

wartość-kontrolna to
1. kontrolka (stała (typu integer, character lub logical) lub wyrażenie typu integer bądź character)
2. : kontrolka
3. kontrolka :
4. kontrolka : kontrolka

Kroki wykonania:

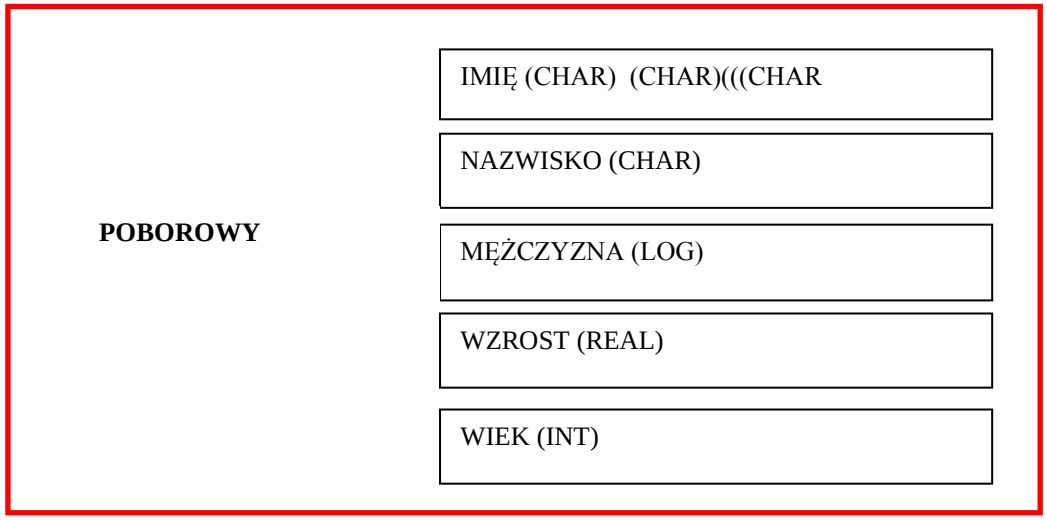
1. Wyznaczenie wartości wyrażenia kontrolnego nazywanej indeksem instrukcji **CASE**.
2. Odszukanie tego zbioru wartości-kontrolnych, w którego zakres wartości wpada indeks instrukcji, wykonanie odpowiedniego bloku instrukcji.
3. Opcjonalnie wykonanie bloku instrukcji **CASE DEFAULT**, jeśli istnieje i nie został wykonany inny blok instrukcji.

```
PROGRAM FOO8
  IMPLICIT NONE
  INTEGER :: A, B
  WRITE (*,'(A15)') 'PODAJ A I B:'
  READ (*,*) A, B

  SELECT CASE (A*B)
    CASE (:20)
       WRITE (*,*) 'MNIEJ NIZ 21'
    CASE (21)
       WRITE (*,*) 'DOKLADNIE 21'
    CASE DEFAULT
       WRITE (*,*) 'ZA DUZO'
    END SELECT

  STOP
END PROGRAM FOO8
```

Typy wartości kontrolnych muszą być takie same jak typ indeksu.
Zbiory wartości-kontrolnych muszą być rozłączne!

TYPY POCHODNE DANYCH

Typy pochodne danych są to tworzone przez programistę (w razie potrzeby) nowe typy zmiennych o zadanej przez programującego wewnętrznej, złożonej "strukturze", składające się z dowolnie wielu zmiennych i struktur zmiennych typów wcześniej zdefiniowanych (“wrodzonych” lub zdefiniowanych wcześniej przez programującego).



Zmienna typu “pochodnego” składa się z wielu uszeregowanych składników, dlatego nazywana jest **strukturą**.

Uwaga: należy dokładnie zrozumieć różnicę pomiędzy TYPEM_ZMIENNEJ a ZMIENNĄ_DANEGO_TYPU.
Np. mamy typ zmiennej real i zmienną A typu real.

Dwa kroki:
1. definiowanie nowych typów pochodnych (można ich zadeklarować dowolnie wiele).
2. deklarowanie nowych zmiennych jako zmiennych danego typu pochodnego.

Instrukcja TYPE

1. Definiowanie typów pochodnych

Instrukcja **TYPE** służy po pierwsze do definiowania nowych typów pochodnych. Nowy typ zmiennych, o złożonej strukturze, składa się z dowolnie wielu zmiennych typów wcześniej "predefiniowanych" lub zdefiniowanych przez programistę.

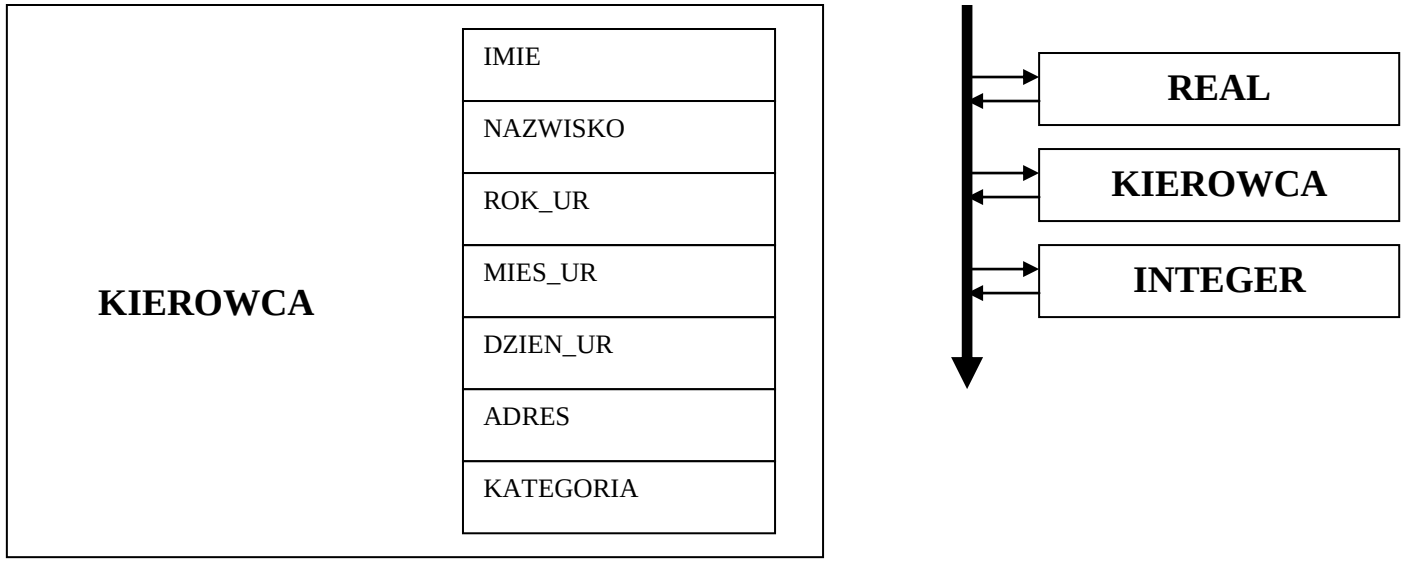
Deklarujemy nazwę typu pochodnego oraz typy i nazwy jego składników:

```
TYPE :: nazwa_typu_zmiennej_strukturalnej
        deklaracja 1-go typu elementów
        deklaracja 2-go typu elementów
        ...
        deklaracja n-tego elementu
END TYPE nazwa_typu_zmiennej_strukturalnej
```

```
TYPE :: SAMOCHOD
   CHARACTER (LEN=20)  :: MARKA,KATEGORIA
   INTEGER             :: ILOSC_MIEJSC
   REAL                :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD

TYPE :: KIEROWCA
   CHARACTER (LEN=20)      :: IMIE, NAZWISKO
   INTEGER                 :: ROK_UR, MIES_UR, DZIEN_UR
   CHARACTER (LEN=100)     :: ADRES
   CHARACTER (LEN=20)      :: KATEGORIA
END TYPE KIEROWCA
```

Nazwy poszczególnych składników struktury są całkowicie lokalne w tej strukturze, tzn. występująca być może w tym samym segmencie programu zmienna IMIE nie ma nic wspólnego z elementem IMIE typu strukturalnego KIEROWCA.

Uwaga: *nazwa_typu_zmiennej_strukturalnej* jest nazwą lokalną



2. Deklarowanie zmiennych typów pochodnych

Instrukcja **TYPE** deklaruje także zmienne jako będące zmiennymi wybranego typu pochodnego.
Oczywiście, wcześniej musi byś zdefiniowany użyty nowy typ pochodny

Przypomnienie: dotychczas np. deklarowaliśmy zmienne A, B jako np. typu integer poprzez: **INTEGER :: A, B**

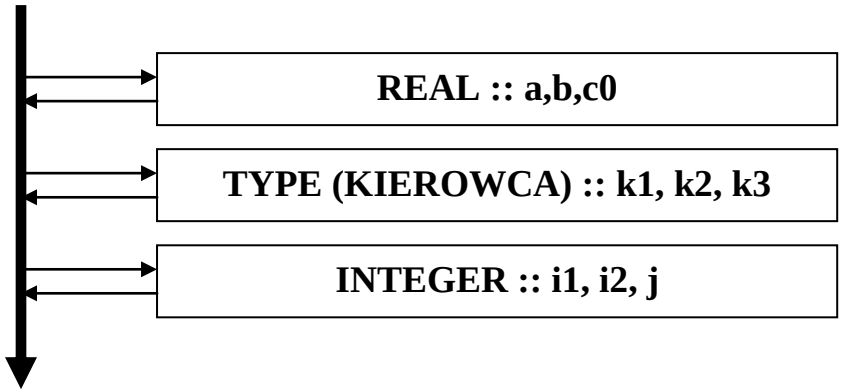
TYPE (*nazwa_typu_zmiennej_strukturalnej*) :: nazwa_zmiennej [, nazwa_zmiennej…]

Np. deklarujemy zmienne JACEK, TOMEK i ADAM jako typu **pochodnego** KIEROWCA:

TYPE (KIEROWCA) :: JACEK, TOMEK, ADAM

Teraz zmienna JACEK jest typu KIEROWCA, czyli jest jedną zmienną, która składa się z wielu elementów: imię, nazwisko, kategoria prawa jazdy itd.

TYPE (SAMOCHOD), DIMENSION(100) :: BAZA_MPK



OGÓLNA ZASADA: Poszczególne elementy zmiennej typu pochodnego (struktury) wskazujemy poprzez podanie tzw. *ścieżki dostępu* do elementu, wymieniając pełną listę zmiennych - od nazwy najbardziej zewnętrznej aż do konkretnego, wskazywanego elementu - rozdzielonych znakiem %

```
TYPE :: KIEROWCA
    CHARACTER (LEN=20) :: IMIE, NAZWISKO
    INTEGER :: ROK_UR, MIES_UR, DZIEN_UR
    CHARACTER (LEN=100) :: ADRES
    CHARACTER (LEN=20) :: KATEGORIA
END TYPE KIEROWCA

TYPE (KIEROWCA) :: JACEK
```

wskazanie elementu KATEGORIA zmiennej strukturalnej JACEK:

jacek%kategoria

```
PROGRAM FOO
IMPLICIT NONE
TYPE :: IM_NAZ
    CHARACTER (LEN=20) :: IMIE, NAZWISKO
END TYPE IM_NAZ
TYPE (IM_NAZ) :: K1,K2,K3
      K1%IMIE='Jacek'
      K1%NAZWISKO='Placek'
STOP
END PROGRAM FOO
```

Gdy element zmiennej typu strukturalnego występuje w wyrażeniu, to typ tego elementu jest taki, jak jego typ podany w definicji zmiennej strukturalnej (a więc w instrukcji TYPE).

Nadawanie wartości początkowych zmiennym typów pochodnych:

przypisanie wartości do elementu zmiennej strukturalnej:

jacek%kategoria=*wartość*

"Konstruktor" struktury (zmiennej typu pochodnego)

Konstruktorem struktury jest struktura bez nazwy:

NAZWA_TYPU_POCHODNEGO (LISTA WYRAŻEŃ)

Liczba i typy wyrażeń na liście wyrażeń musi być zgodna z liczbą i typami komponentów typu pochodnego. Jeśli jednak jest to niezbędne, kompilator dokona (w miarę możliwości) konwersji podstawowych (wrodzonych) typów zmiennych.

```
A1=SAMOCHOD ('AUDI',30.3, 80.5)
WRITE (*,'(1X,A10,2I6)') A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
```

```
PROGRAM FOO
IMPLICIT NONE
TYPE :: SAMOCHOD
    CHARACTER (LEN=20) :: MARKA
    INTEGER :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD
TYPE (SAMOCHOD) :: A1
  A1%MARKA='MAN'
  A1%LADOWNOSC=40
  A1%ZASIEG=500
  WRITE (*,'(1X,A10,2I6)') A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
  A1=SAMOCHOD ('OPEL',30, 800)
  WRITE (*,'(1X,A10,2I6)') A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
STOP
END PROGRAM FOO
```

Błędna konstrukcja:

```
A1=SAMOCHOD ('STAR',30)
WRITE (*,'(1X,A10,2I6)') A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
```

W instrukcji TYPE można zastosować instrukcję TYPE dla zadeklarowania elementów jako mających wcześniej zdefiniowany typ pochodny rekordowy czyli wykorzystując wcześniej zdefiniowane typy strukturalne. W ten sposób zmienna strukturalna może mieć wielopoziomową strukturę.

```
PROGRAM FOO
IMPLICIT NONE

TYPE :: SAMOCHOD
    CHARACTER (LEN=20) :: MARKA
    INTEGER :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD
TYPE :: NACZEPA
    CHARACTER (LEN=20) :: MARKA
    INTEGER :: LADOWNOSC
    REAL :: DLUGOSC,SZEROKOSC, WYSOKOSC
END TYPE NACZEPA

TYPE :: TIR
    TYPE (SAMOCHOD) :: CIAGNIK
    TYPE (NACZEPA) :: SKRZYNIA
END TYPE TIR

TYPE (TIR) :: C1, C2

C1%CIAGNIK%ZASIEG=540
C2%SKRZYNIA%LADOWNOSC=30
WRITE (*,'(1X,A10,I6)') 'ZASIEG: ',C1%CIAGNIK%ZASIEG
WRITE (*,'(1X,A10,I6)') 'LADOWNOSC: ',C2%SKRZYNIA%LADOWNOSC
STOP
END PROGRAM FOO
```

A teraz to samo, ale z wykorzystaniem modułu:

```
MODULE DEFS
IMPLICIT NONE

TYPE :: SAMOCHOD
   CHARACTER (LEN=20) :: MARKA
   INTEGER :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD

TYPE :: NACZEPA
   CHARACTER (LEN=20) :: MARKA
   INTEGER :: LADOWNOSC
   REAL    ::    DLUGOSC,SZEROKOSC,
WYSOKOSC
END TYPE NACZEPA

TYPE :: TIR
   TYPE (SAMOCHOD) :: CIAGNIK
   TYPE (NACZEPA) :: SKRZYNIA
END TYPE TIR\

END MODULE DEFS
```

```
PROGRAM FOO
USE DEFS
IMPLICIT NONE
TYPE (TIR) :: C1, C2
   C1%CIAGNIK%ZASIEG=540
   C2%SKRZYNIA%LADOWNOSC=30
   WRITE              (*,'(1X,A10,I6)')              'ZASIEG:
',C1%CIAGNIK%ZASIEG
   WRITE (*,'(1X,A10,I6)') 'LADOWNOSC:',&
                %SKRZYNIA%LADOWNOSC
STOP
END PROGRAM FOO
```

Poszczególne elementy zmiennych typu strukturalnego mogą być zapisywane do zbiorów lub z nich czytane, ale próba zapisu/odczytu zmiennej strukturalnej jako całości powoduje powstanie błędu.

```
PROGRAM FOO
IMPLICIT NONE
TYPE :: SAMOCHOD
    CHARACTER (LEN=20) :: MARKA
    INTEGER :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD
TYPE (SAMOCHOD) :: A1
READ (*,*) A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
WRITE (*,'(1X,A10,2I6)') A1%MARKA,A1%LADOWNOSC,A1%ZASIEG
STOP
END PROGRAM FOO
```

Błędny jest natomiast zapis:

```
READ (*,*) A1
WRITE (*,*) A1
```

Tablice zmiennych pochodnych

```
PROGRAM FOO
IMPLICIT NONE
TYPE :: SAMOCHOD
    CHARACTER (LEN=20) :: MARKA
    INTEGER :: LADOWNOSC, ZASIEG
END TYPE SAMOCHOD
TYPE  (SAMOCHOD), DIMENSION(100) :: BAZA
INTEGER :: I
BAZA(1)%MARKA='MAN'
BAZA(21)%LADOWNOSC=120
DO I=31,51
  BAZA(I)%ZASIEG=100
END DO
DO I=50,100
  BAZA(I)=SAMOCHOD('IKARUS',10,300)
END DO
WRITE (*,'(1X,A10,2I6)') BAZA(1)%MARKA,BAZA(21)%LADOWNOSC,BAZA(99)%ZASIEG
STOP
END PROGRAM FOO
```

PRZECIĄŻANIE OPERATORÓW

Operatory zaimplementowane w kompilartorze (np. -, *) mogą być przeciążane w celu umożliwienia zastosowania ich do innych typów danych niż to pierwotnie zaimplementowano w kompilatorze.

Sposób przeciążenia definiujemy/opisujemy/ w odpowiednim module.

1. Określamy ogólny symbol operatora (instrukcja **INTERFACE OPERATOR**)
2. Deklarujemy procedury modułu opisujące działanie operatora (instr. **MODULE PROCEDURE**)

Każdy operator zaimplementowany w kompilatorze może być przeciążony. Operatory, które mogą być jedno- lub dwuargumentowe (np. operator +), jeśli oba przypadki mają być przeciążone, to oba muszą być osobno zdefiniowane.

Instrukcja MODULE PROCEDURE

Instrukcja MODULE PROCEDURE ustala, że nazwy występujące w jej liście nazw są składnikami nazwy ogólnej przeciążonego operatora

MODULE PROCEDURE lista_nazw

Instrukcja ta może występować tylko w ramach interface'u operatora w module lub w segmencie programu który otrzymuje dostęp do modułu poprzez użycie instrukcji USE.

Np.:

```
INTERFACE OPERATOR (+)
   MODULE PROCEDURE SUM_C, SUM_B
END INTERFACE
```

Procedury te mogą mieć jeden lub dwa argumenty (odpowiednio lewo- i prawo-stronny) z atrybutem INTENT(IN), odpowiednio dla operatorów jedno i dwuargumentowych. Wybór odpowiedniej procedury odbywa się na podstawie analizy liczby, typu, rank'u i kind'u argumentów, więc definicje procedur muszą być jednoznaczne.

Przeciążenie operatora nie zmienia jego kolejności wykonywania.

```
MODULE FOO_SUM

  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.14

  TYPE :: CIRCLE
     REAL :: X,Y,R
  END TYPE CIRCLE
  TYPE :: BOX
     REAL :: X1,Y1,X2,Y2
  END TYPE BOX
  INTERFACE OPERATOR (+)
      MODULE PROCEDURE SUM_C, SUM_B
  END INTERFACE
  CONTAINS
  FUNCTION SUM_C (A, B)
    TYPE(CIRCLE), INTENT(IN) :: A, B
    REAL :: SUM_C
    SUM_C=PI*(A%R**2+B%R**2)
    RETURN
  END FUNCTION SUM_C
  FUNCTION SUM_B (A, B)
     TYPE(BOX), INTENT(IN) :: A, B
     REAL :: SUM_B
     SUM_B=(A%X2-A%X1)*(A%Y2-A%Y1)+(B%X2-&
                B%X1)*(B%Y2-B%Y1)
     RETURN
   END FUNCTION SUM_B

END MODULE FOO_SUM
```

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
        C1%R=5.
        C2%R=10.
        B1%X2=5.
        B1%Y2=5.
        B2%X2=15.
        B2%Y2=15.
        WRITE(*,*) I+5
        WRITE(*,*) C1+C2
        WRITE(*,*) B1+B2
STOP
END PROGRAM FOO
```

```
!Uwaga!
!C=2.71**2+(C1+C2)     OK, nasz + “wie” jak dodać
!struktury C1 i C2 (wynikiem jest skalar)
!C=2.71**2+C1+C2       ŹLE, nasz + “nie wie” jak
!dodać skalar do struktury
```

Możliwe jest zastosowanie zarówno operatora + jak i procedur SUM_B, SUM_C (ale dla odpowiednich typów argumentów!):

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
C1%R=5.
C2%R=10.
B1%X2=5.
B1%Y2=5.
B2%X2=15.
B2%Y2=15.
WRITE(*,*) I+5
WRITE(*,*) C1+C2
WRITE(*,*) B1+B2
WRITE(*,*) SUM_B(B1,B2) ! możliwe ale nieelganckie !!!!
!!! WRITE(*,*) SUM_C(B1,B2)   ! bład: niezgodnosc typów arg. formalnych i aktualnych
STOP
END PROGRAM FOO
```

Lepiej jest jednak ograniczny ć “widoczność” procedur SUM_B, SUM_C:

```
MODULE FOO_SUM
  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.14159
  TYPE :: CIRCLE
     REAL :: X,Y,R
  END TYPE CIRCLE
  TYPE :: BOX
     REAL :: X1,Y1,X2,Y2
  END TYPE BOX
  INTERFACE OPERATOR (+)
     MODULE PROCEDURE SUM_C, SUM_B
  END INTERFACE
  PRIVATE :: SUM_C, SUM_B
  CONTAINS
  FUNCTION SUM_C (A, B)
    TYPE(CIRCLE), INTENT(IN) :: A, B
    REAL :: SUM_C
    SUM_C=PI*(A%R**2+B%R**2)
    RETURN
  END FUNCTION SUM_C
  FUNCTION SUM_B (A, B)
     TYPE(BOX), INTENT(IN) :: A, B
     REAL :: SUM_B
     SUM_B=(A%X2-A%X1)*(A%Y2-A%Y1)+(B%X2-
B%X1)*(B%Y2-B%Y1)
     RETURN
   END FUNCTION SUM_B
END MODULE FOO_SUM
```

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
C1%R=5.
C2%R=10.
B1%X2=5.
B1%Y2=5.
B2%X2=15.
B2%Y2=15.
WRITE(*,*) I+5
WRITE(*,*) C1+C2
WRITE(*,*) B1+B2
! WRITE(*,*) SUM_B(B1,B2)      blad
! WRITE(*,*) SUM_C(c1,c2)      blad
STOP
END PROGRAM FOO
```

Teraz nie jest możliwe zastosowanie procedur SUM_B i SUM_C. “Widać” tylko przeciążony operator +.

DEFINIOWANIE NOWYCH OPERATORÓW

Operatory definiowane przez programistę mają postać:

. nazwa .

nazwa może się składać tylko z liter i musi byc różna od naze operatorów zaimplementowanych.

Tak jak przy definicji przeciążenia operatrów, muszą być jeden lub dwa argumenty (odpowiednio lewo- i prawo-stronny) z atrybutem INTENT(IN), odpowiednio dla operatorów jedno i dwuargumentowych.

```
MODULE kon_op
 INTERFACE OPERATOR (.kon.)
    MODULE PROCEDURE kon_jed, kon_dwa
 END INTERFACE
 CONTAINS
  FUNCTION kon_jed (s)
   character(len=2) :: kon_jed
   character(len=1), INTENT(IN) :: s
     kon_jed=’_‘//s
     return
  END FUNCTION kon_jed
 FUNCTION kon_dwa (s1, s2)
  character(len=2) :: kon_dwa
  character(len=1), INTENT(IN) :: s1, s2
    kon_dwa=s1//s2
    return
  END FUNCTION kon_dwa
END MODULE kon_op
```

```
PROGRAM FOO
USE kon_op
IMPLICIT NONE
character(len=1) :: s1='a', s2='b'
WRITE(*,'(1x,a)') .kon.s1
WRITE(*,'(1x,a)') s1.kon.s2
WRITE(*,'(1x,a)') kon_jed(s1)  !!! nieelegancko!!!
STOP
END PROGRAM FOO
```

```
MODULE kon_op
 INTERFACE OPERATOR (.kon.)
    MODULE PROCEDURE kon_jed, kon_dwa
 END INTERFACE
 PRIOVATE KON_JED, KON_DWA
 CONTAINS
  FUNCTION kon_jed (s)
   character(len=2) :: kon_jed
   character(len=1), INTENT(IN) :: s
    kon_jed=’_‘//s
    return
  END FUNCTION kon_jed
 FUNCTION kon_dwa (s1, s2)
  character(len=2) :: kon_dwa
  character(len=1), INTENT(IN) :: s1, s2
    kon_dwa=s1//s2
    return
  END FUNCTION kon_dwa
END MODULE kon_op
```

```
PROGRAM FOO
USE kon_op
IMPLICIT NONE
character(len=1) :: s1='a', s2='b'
WRITE(*,'(1x,a)') .kon.s1
WRITE(*,'(1x,a)') s1.kon.s2
WRITE(*,'(1x,a)') kon_jed(s1)  !!! nieelegancko!!!
STOP
END PROGRAM FOO
```

Programisty jednoargumentowy	najwcześniej	.INV. A
**		10**4
* lub /		5*4
Jednoargumentowy + lub -		-5
Dwuargumentowy + lub -		5+5
//		STR1//SRT2
.GT. > .LE. <= etc.		A > B
.NOT.		.NOT.CON
.AND.		A.AND.B
.OR.		A.OR.B
.EQV. lub .NEQV.		A.EQV.B
Programisty dwuargumentowy	najpóźniej	X.MIN.Y

PRZYPISANIE (=) DEFINIOWANE PRZEZ PROGRAMISTĘ

Przypisanie pomiędzy dwoma obiektami różnych typów pochodnych lub pomiędzy obiektem typu pochodnego i zaimplementowanego musi być explicite zdefiniowane przez programistę.
Przeciążenie operatora przypisania (=), podobne do przeciążania innych operatorów, dokonywane jest za pomocą

Zdefiniowania odpowiedniego podprogramu (subroutine) z:

1. pierwszym argumentem: obiektem (zmienną) który “otrzymuje” wartość, musi mieć atrybut INTENT(OUT);
2. drugim argumentem: wyrażeniem, którego wartość jest przekształcana i przypisywana pierwszemu argumentowi, musi mieć atrybut INTENT(IN).

```
MODULE pds_op
IMPLICIT NONE
TYPE :: DZIEN
  INTEGER :: D
  CHARACTER(LEN=10) :: M
END TYPE DZIEN
  INTERFACE ASSIGNMENT (=)
     MODULE PROCEDURE DD, MM
  END INTERFACE
  private :: DD, MM
  CONTAINS
  SUBROUTINE DD (A, B)
    TYPE (DZIEN), INTENT(OUT) :: A
    INTEGER, INTENT(IN) :: B
      A%D=B
      return
   END SUBROUTINE DD
  SUBROUTINE MM (A, B)
    TYPE (DZIEN), INTENT(OUT) :: A
    CHARACTER(LEN=10), INTENT(IN) :: B
      A%m=B
      return
   END SUBROUTINE MM
END MODULE PDS_op
```

```
PROGRAM FOO
USE PDS_op
IMPLICIT NONE
TYPE (DZIEN) :: AA
CHARACTER(LEN=10) :: BB
BB='JULY'
AA=BB
AA=11
WRITE(*,'(I3,X,A10)') AA%D,AA%M
STOP
END PROGRAM FOO
```

Każdy z podprogramów musi zawierać przypisanie wartości pierwszemu argumentowi.

ROZSZERZANIE SEMANTYKI JĘZYKA FORTRAN F90

```
MODULE circle_class
  PRIVATE

  TYPE :: circle
    PRIVATE
    REAL :: x,y,r
  END TYPE circle

  INTERFACE new
    MODULE PROCEDURE new_cir
  END INTERFACE
  INTERFACE draw
    MODULE PROCEDURE draw_cir
  END INTERFACE
  PUBLIC :: circle, new, draw

  CONTAINS
  subroutine draw_cir (o)
    TYPE(circle), intent(in) :: o
    WRITE(*,*) ' Rys kolo: ', o%x, o%y
    return
  END SUBROUTINE draw_cir

  SUBROUTINE new_cir(o,x,y,r)
    TYPE(circle), intent(out) :: o
    REAL, intent(in)       :: x,y,r
    o%x = x
    o%y = y
    o%r = r
    return
  END SUBROUTINE new_cir
END MODULE circle_class
```

```
PROGRAM FOO
USE circle_class
IMPLICIT NONE
TYPE (CIRCLE) :: C1
call new(c1,5.,5.,6.)
call draw(c1)
STOP
END PROGRAM FOO
```

W zasadzie, klasa to typ posiadajacy PRYWATNE (PRIVATE) składniki wraz z konstruktorami obiektów oraz estawem, metod umożliwiających różne operacje na obiektach danej klasy.

Używanie nazw ogólnych (*generic*) dla operacji takich jak np. *draw*, *new* itd. Pozwala nam używać tych samych nazw dla metod innych klas i dziedziczenie typów i procedur z innych klas.

```
MODULE box_class

  PRIVATE
  TYPE :: box
   PRIVATE
   REAL :: x1,y1,x2,y2
  END TYPE box

  INTERFACE new
   MODULE PROCEDURE new_box
  END INTERFACE
  INTERFACE draw
   MODULE PROCEDURE draw_box
  END INTERFACE

  PUBLIC :: box, new, draw

  CONTAINS
  subroutine draw_box (o)
   TYPE(box), intent(in) :: o
   WRITE(*,*) ' Rys box:', o%x1, o%y1, o%x2,
o%y2
   return
  END SUBROUTINE draw_box
  SUBROUTINE new_box(o,x1,y1,x2,y2)
   TYPE(box), intent(out) :: o
   REAL, intent(in)       :: x1,y1,x2,y2
   o%x1 = x1
   o%y1 = y1
   o%x2 = x2
   o%y2 = y2
   return
  END SUBROUTINE new_box
END MODULE box_class
```

Instrukcja DATA

Nadaje wartości początkowe lokalnym zmiennym i tablicom.

DATA lista_zmiennych / lista_wartości / [,lista_zmiennych / lista_wartości /]

gdzie:

- **lista_zmiennych**: lista nazw zmiennych, tablic, elementów tablic, elementów struktur oraz listy DO-implikowanego.
- **lista_wartości**: lista stałych, stałych zwielokrotnionych, łańcuchów, oddzielonych przecinkami.
  stała zwielokrotniona: n*const (3*5 = 5,5,5)
- **ilość elementów** listy_zmiennych musi być równa ilości elementów listy_wartości;
  (nazwa tablicy => tyle wartości ile elementów liczy tablica).

1. Nadanie wartości zmiennym skalarnym i tablicom

```
REAL :: A,B,I,J
DATA A,B,I,J /1.5,1.6,1.7,1.8/
! A=1.5, B=1.6, I=1, J=1

REAL :: A(5)
INTEGER :: I,J,K
DATA I,J,K,A /3*1, 2.3, 4.5, 3*1.5/
! I=1, J=1, K=1, A(1)=2.3, A(2)=4.5, A(3) do A(5)=1.5
```

2. Nadawanie warości zmiennym znakowym

```
CHARACTER(LEN=5) :: S1
DATA S1 /'PUDELKO'/
! s1='pudel'
```

3. Nadawanie wartości tablicom za pomocą wewnętrznej pętli DO-implikowanego

```
REAL :: X(15,15)
DATA (( X(I,J),  I=1,15),  J=1,15 ),A,B,C  /15*1.0, 15*2.0, 198*3.0/
! x(*,1)=1.0,  x(*,2)=2.0,  x(*,3÷15)=3.0, a=3.0, b=3.0, c=3.0
```

4. Zmienne typów pochodnych

```
TYPE POZ
      REAL :: X, Y
      INTEGER :: Z
END TYPE POZ


TYPE (POZ) :: STAT1, STAT2
DATA STAT1 / POZ(3., 5., 12)/, STAT2%Z /21/
```

Nadanie elementowi **Z** struktury **STAT2** wartości 21

Nadanie wartości wszystkim elementom struktury **STAT1** za pomocą konstruktora

KOLEJNOŚĆ WYKONYWANIA OPERATORÓW

Programisty jednoargumentowy	najwcześniej	.INV. A
**		10**4
* lub /		5*4
Jednoargumentowy + lub -		-5
Dwuargumentowy + lub -		5+5
//		STR1//SRT2
.GT. > .LE. <= etc.		A > B
.NOT.		.NOT.CON
.AND.		A.AND.B
.OR.		A.OR.B
.EQV. lub .NEQV.		A.EQV.B
Programisty dwuargumentowy	najpóźniej	X.MIN.Y

W wyrażeniach zapisanych bez nawiasów, operatory są wiązane ze swymi argumentami w kolejności „od góry do dołu tabeli”. Dla operatorów o jednakowej kolejności obowiązuje kolejność „od strony lewej do prawej” wyrazenia, za wyjątkiem operatora potęgi (tu obowiązuje kolejność „od prawej do lewej”!).

Kolejność operatorów zaimplementowanych jest stała i nie może ulec zmianie.

PRZECIĄŻANIE OPERATORÓW

Operatory zaimplementowane w kompilartorze (np. -, *) mogą być przeciążane w celu umożliwienia zastosowania ich do innych typów danych niż to pierwotnie zaimplementowano w kompilatorze.

Sposób przeciążenia zazwyczaj definujemy w odpowiednim module.

3. Określamy ogólny symbol operatora (instrukcja **INTERFACE OPERATOR**)
4. Deklarujemy procedury modułu opisujące działanie operatora (instr. **MODULE PROCEDURE**)

Każdy operator zaimplementowany w kompilatorze może być przeciążony. Operatory, które mogą być jedno- lub dwuargumentowe (np. +), jeśli oba przypadki mają być przeciążone, to oba muszą być osobno zdefiniowane.

Instrukcja INTERFACE OPARATOR

Instrukcja INTERFACE OPERATOR tworzy nagłówek bloku interfejsu dla funkcji opisujących rozszerzenie działania operatora wewnetrznego.

```
INTERFACE OPERATOR (int_operator)
    MODULE PROCEDURE proc_names
END INTERFACE
```

Procedury/funkcje wyspecyfikowane w bloku interface muszą mieć tą samą (jeden lub dwa) liczbę parametrów co przeciążany operator. Parametry proc/funkc nie mogą być parametrami opcjonalnymi i muszą być deklarowane z atrubutem INTENT (IN).

Np.:
```
INTERFACE OPERATOR (+)
    MODULE PROCEDURE SUM_C, SUM_B
END INTERFACE
```

Instrukcja MODULE PROCEDURE

Instrukcja MODULE PROCEDURE ustala, że nazwy występujące w jej liście nazw są składnikami nazwy ogólnej przeciążonego operatora

MODULE PROCEDURE lista_nazw

Instrukcja ta może występować tylko w ramach interface'u operatora w module lub w segmencie programu który otrzymuje dostęp do modułu poprzez użycie instrukcji USE.

Procedury te mogą mieć jeden lub dwa argumenty (odpowiednio lewo- i prawo-stronny) z atrybutem INTENT(IN), odpowiednio dla operatorów jedno i dwuargumentowych. Wybór odpowiedniej procedury odbywa się na podstawie analizy liczby, typu, rank'u i kind'u argumentów, więc definicje procedur muszą być jednoznaczne.

Przeciążenie operatora nie zmienia jego kolejności wykonywania.

Np.: **INTERFACE OPERATOR (+)**
MODULE PROCEDURE SUM_C, SUM_B
END INTERFACE

```
MODULE FOO_SUM
  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.14159

  TYPE :: CIRCLE
     REAL :: X,Y,R
  END TYPE CIRCLE
  TYPE :: BOX
     REAL :: X1,Y1,X2,Y2
  END TYPE BOX
  INTERFACE OPERATOR (+)
     MODULE PROCEDURE SUM_C, SUM_B
  END INTERFACE
  CONTAINS
  FUNCTION SUM_C (A, B)
    TYPE(CIRCLE), INTENT(IN) :: A, B
    REAL :: SUM_C
    SUM_C=PI*(A%R**2+B%R**2)
    RETURN
  END FUNCTION SUM_C
  FUNCTION SUM_B (A, B)
     TYPE(BOX), INTENT(IN) :: A, B
     REAL :: SUM_B
     SUM_B=(A%X2-A%X1)*(A%Y2-A%Y1)+(B%X2-&
                B%X1)*(B%Y2-B%Y1)
     RETURN
   END FUNCTION SUM_B
END MODULE FOO_SUM
```

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
      C1%R=5.
      C2%R=10.
      B1%X2=5.
      B1%Y2=5.
      B2%X2=15.
      B2%Y2=15.
      WRITE(*,*) I+5
      WRITE(*,*) C1+C2
      WRITE(*,*) B1+B2
STOP
END PROGRAM FOO
```

Uwaga!

C=2.712+(C1+C2)**
C=2.712+C1+C2** ! + "nie wie" jak dodać skalar do struktury

Możliwe jest zastosowanie zarówno operatora + jak i procedur SUM_B, SUM_C (ale dla odpowiednich typów argumentów!):

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
C1%R=5.
C2%R=10.
B1%X2=5.
B1%Y2=5.
B2%X2=15.
B2%Y2=15.
WRITE(*,*) I+5
WRITE(*,*) C1+C2
WRITE(*,*) B1+B2
WRITE(*,*) SUM_B(B1,B2) ! możliwe ale nieelganckie !!!!
!!! WRITE(*,*) SUM_C(B1,B2)   ! blad: niezgodnosc typów arg. formalnych i aktualnych
STOP
END PROGRAM FOO
```

Lepiej jest jednak ograniczny ć "widoczność" procedur SUM_B, SUM_C:

```
MODULE FOO_SUM
  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.14159
  TYPE :: CIRCLE
     REAL :: X,Y,R
  END TYPE CIRCLE
  TYPE :: BOX
     REAL :: X1,Y1,X2,Y2
  END TYPE BOX
  INTERFACE OPERATOR (+)
      MODULE PROCEDURE SUM_C, SUM_B
  END INTERFACE
  PRIVATE :: SUM_C, SUM_B
  CONTAINS
  FUNCTION SUM_C (A, B)
    TYPE(CIRCLE), INTENT(IN) :: A, B
    REAL :: SUM_C
    SUM_C=PI*(A%R**2+B%R**2)
    RETURN
  END FUNCTION SUM_C
  FUNCTION SUM_B (A, B)
     TYPE(BOX), INTENT(IN) :: A, B
     REAL :: SUM_B
     SUM_B=(A%X2-A%X1)*(A%Y2-A%Y1)+(B%X2-
B%X1)*(B%Y2-B%Y1)
     RETURN
   END FUNCTION SUM_B
END MODULE FOO_SUM
```

```
PROGRAM FOO
USE FOO_SUM
IMPLICIT NONE
INTEGER :: I = 90
TYPE (CIRCLE) :: C1, C2
TYPE (BOX) :: B1, B2
C1%R=5.
C2%R=10.
B1%X2=5.
B1%Y2=5.
B2%X2=15.
B2%Y2=15.
WRITE(*,*) I+5
WRITE(*,*) C1+C2
WRITE(*,*) B1+B2
! WRITE(*,*) SUM_B(B1,B2)      blad
! WRITE(*,*) SUM_C(c1,c2)      blad
STOP
END PROGRAM FOO
```

Teraz nie jest możliwe zastosowanie procedur SUM_B i SUM_C. “Widac tylko przeciążony operator +.

PRZECIĄŻANIE FUNKCJI I PROCEDUR

Przeciążenie nazwy funkcji lub procedury pozwala wywoływać poprzez jedną nazwę różne funkcje (procedury) np. dopasowane do typów argumentów

Uwaga: wiele funkcji wewnętrznych ma tą cechę ☺ a więc jest to zjawisko dobrze nam znane. Funkcje i procedury przeciążoN zawsze wymagają interfejsu.

Sposób przeciążenia zazwyczaj definujemy w odpowiednim module.

1. Określamy ogólny symbol operatora (instrukcja **INTERFACE**)
2. Deklarujemy procedury modułu opisujące działanie operatora (instr. **MODULE PROCEDURE**)

Instrukcja INTERFACE

Instrukcja INTERFACE tworzy nagłówek bloku interfejsu dla funkcji opisujących nowe rozszerzenie działania funkcji/procedury.

```
INTERFACE nazwa_funkcji
    MODULE PROCEDURE aux_proc_names
END INTERFACE
```

Np.:
```
INTERFACE max
    MODULE PROCEDURE max_foo1, max_foo2
END INTERFACE
```

```
MODULE FOOMOD
  IMPLICIT NONE
  REAL, PARAMETER :: PI=3.14159
  TYPE :: CIRCLE
     REAL :: X,Y,R
  END TYPE CIRCLE
  TYPE :: BOX
     REAL :: X1,Y1,X2,Y2
  END TYPE BOX
  INTERFACE MAX
     MODULE PROCEDURE B_CIR, B_BOX
  END INTERFACE
  PRIVATE :: B_CIR, B_BOX
  CONTAINS
  FUNCTION B_CIR (A, B)
    TYPE(CIRCLE), INTENT(IN) :: A, B
    REAL :: B_CIR, P1, P2
    P1=PI*A%R**2
    P2=PI*B%R**2
    B_CIR=MAX(P1,P2)
    RETURN
  END FUNCTION B_CIR
  FUNCTION B_BOX (A, B)
     TYPE(BOX), INTENT(IN) :: A, B
     REAL :: B_BOX, P1, P2
     P1=ABS(A%X2-A%X1)*ABS(A%Y2-A%Y1)
     P2=ABS(B%X2-B%X1)*ABS(B%Y2-B%Y1)
     B_BOX=MAX(P1,P2)
     RETURN
   END FUNCTION B_BOX
END MODULE FOOMOD
```

```
PROGRAM FOO
      USE FOOMOD
      IMPLICIT NONE
      TYPE (CIRCLE) :: C1, C2
      TYPE (BOX) :: B1, B2
      C1%R=5.
      C2%R=10.
      B1%X2=5.
      B1%Y2=5.
      B2%X2=15.
      B2%Y2=15.
      WRITE(*,*) MAX(5,10)
      WRITE(*,*) MAX(2.5,3.6)
      WRITE(*,*) MAX(C1,C2)
      WRITE(*,*) MAX(B1,B2)
      STOP
END PROGRAM FOO
```

DEFINIOWANIE NOWYCH OPERATORÓW

Nowe oparatory pozwalają tworzyć własne poręczne narzędzia a także są szczególnie uzyteczne przy definiowaniu całkiem nowych operacji na strukturach. Operatory definiowane przez programistę mają postać:

. nazwa .

nazwa może się składać tylko z liter i musi być różna od nazw operatorów zaimplementowanych.

Tak jak przy definicji przeciążenia operatrów, muszą być jeden lub dwa argumenty (odpowiednio lewo- i prawo-stronny) z atrybutem INTENT(IN), odpowiednio dla operatorów jedno i dwuargumentowych.

```
MODULE kon_op1
implicit none
 INTERFACE OPERATOR (.alt.)
    MODULE PROCEDURE a
 END INTERFACE
 PRIVATE :: a
 CONTAINS
  FUNCTION a (s)
   character(len=1) :: a
   character(len=1), INTENT(IN) :: s
    select case (s)
      case ('a':'z')
        a=char(iachar(s)-32)
      case ('A':'Z')
        a=char(iachar(s)+32)
    end select
    return
  END FUNCTION a
END MODULE kon_op1
```

```
PROGRAM FOO
USE kon_op1
IMPLICIT NONE
character(len=1) :: s1='a', s2='C'
WRITE(*,'(1x,25a)') .alt.s1
WRITE(*,'(1x,25a)') .alt.s2
STOP
END PROGRAM FOO
```

```
MODULE kon_op2
 INTERFACE OPERATOR (.kon.)
    MODULE PROCEDURE kon_jed, kon_dwa
 END INTERFACE
 PRIVATE :: KON_JED, KON_DWA
 CONTAINS
  FUNCTION kon_jed (s)
   character(len=2) :: kon_jed
   character(len=1), INTENT(IN) :: s
    kon_jed=’_‘//s
    return
  END FUNCTION kon_jed
 FUNCTION kon_dwa (s1, s2)
  character(len=2) :: kon_dwa
  character(len=1), INTENT(IN) :: s1, s2
   kon_dwa=s1//s2
   return
  END FUNCTION kon_dwa
END MODULE kon_op2
```

```
PROGRAM FOO
USE kon_op2
IMPLICIT NONE
character(len=1) :: s1='a', s2='b'
WRITE(*,'(1x,a)') .kon.s1
WRITE(*,'(1x,a)') s1.kon.s2
STOP
END PROGRAM FOO
```

Programisty jednoargumentowy	najwcześniej	.INV. A
**		10**4
* lub /		5*4
Jednoargumentowy + lub -		-5
Dwuargumentowy + lub -		5+5
//		STR1//SRT2
.GT. > .LE. <= etc.		A > B
.NOT.		.NOT.CON
.AND.		A.AND.B
.OR.		A.OR.B
.EQV. lub .NEQV.		A.EQV.B
Programisty dwuargumentowy	najpóźniej	X.MIN.Y

PRZYPISANIE (=) DEFINIOWANE PRZEZ PROGRAMISTĘ

Przypisanie pomiędzy dwoma obiektami różnych typów pochodnych lub pomiedzy obiektem typu pochodnego i zaimplementowanego musi być explicite zdefiniowane przez programistę.

Przeciążenie operatora przypisania (**=**), podobne do przeciążania innych operatorów, dokonywane jest za pomocą zdefiniowania odpowiedniego **podprogramu (subroutine)** z:

3. pierwszym argumentem: obiektem (zmienną) który “otrzymuje” wartość, musi mieć atrybut INTENT(OUT);
4. drugim argumentem: wyrażeniem, którego wartość jest przekształcana i przypisywana pierwszemu argumentowi, musi mieć atrybut INTENT(IN).

```
MODULE pds_op
IMPLICIT NONE
TYPE :: DZIEN
  INTEGER :: D
  CHARACTER(LEN=10) :: M
END TYPE DZIEN
  INTERFACE ASSIGNMENT (=)
     MODULE PROCEDURE DD, MM
  END INTERFACE
  private :: DD, MM
  CONTAINS
  SUBROUTINE DD (A, B)
    TYPE (DZIEN), INTENT(OUT) :: A
    INTEGER, INTENT(IN) :: B
      A%D=B
      return
   END SUBROUTINE DD
  SUBROUTINE MM (A, B)
    TYPE (DZIEN), INTENT(OUT) :: A
    CHARACTER(LEN=10), INTENT(IN) :: B
      A%m=B
      return
   END SUBROUTINE MM
END MODULE PDS_op
```

```
PROGRAM FOO
USE PDS_op
IMPLICIT NONE
TYPE (DZIEN) :: AA
CHARACTER(LEN=10) :: BB
BB='JULY'
AA=BB
AA=11
WRITE(*,'(I3,X,A10)') AA%D,AA%M
STOP
END PROGRAM FOO
```

Każdy z podprogramów musi zawierać przypisanie wartości pierwszemu argumentowi.

ROZSZERZANIE SEMANTYKI JĘZYKA FORTRAN F90

```
MODULE circle_class
  PRIVATE

  TYPE :: circle
    PRIVATE
    REAL :: x,y,r
  END TYPE circle

  INTERFACE new
    MODULE PROCEDURE new_cir
  END INTERFACE
  INTERFACE draw
    MODULE PROCEDURE draw_cir
  END INTERFACE
  PUBLIC :: circle, new, draw

  CONTAINS
  subroutine draw_cir (o)
    TYPE(circle), intent(in) :: o
    WRITE(*,*) ' Rys kolo: ', o%x, o%y
    return
  END SUBROUTINE draw_cir

  SUBROUTINE new_cir(o,x,y,r)
    TYPE(circle), intent(out) :: o
    REAL, intent(in)       :: x,y,r
    o%x = x
    o%y = y
    o%r = r
    return
  END SUBROUTINE new_cir
END MODULE circle_class
```

```
PROGRAM FOO
USE circle_class
IMPLICIT NONE
TYPE (CIRCLE) :: C1
call new(c1,5.,5.,6.)
call draw(c1)
STOP
END PROGRAM FOO
```

W zasadzie, klasa to typ posiadający PRYWATNE (PRIVATE) składniki wraz z konstruktorami obiektów oraz zestawem, metod umożliwiających różne operacje na obiektach danej klasy.

Używanie nazw ogólnych (*generic*) dla operacji takich jak np. *draw*, *new* itd. Pozwala nam używać tych samych nazw dla metod innych klas i dziedziczenie typów i procedur z innych klas.

```
MODULE box_class

  PRIVATE

  TYPE :: box
   PRIVATE
   REAL :: x1,y1,x2,y2
  END TYPE box

  INTERFACE new
   MODULE PROCEDURE new_box
  END INTERFACE
  INTERFACE draw
   MODULE PROCEDURE draw_box
  END INTERFACE
  PUBLIC :: box, new, draw

  CONTAINS

    subroutine draw_box (o)
     TYPE(box), intent(in) :: o
     WRITE(*,*) ' Rys box:', o%x1, o%y1, o%x2,
  o%y2
     return
    END SUBROUTINE draw_box
    SUBROUTINE new_box(o,x1,y1,x2,y2)
     TYPE(box), intent(out) :: o
     REAL, intent(in)       :: x1,y1,x2,y2
     o%x1 = x1
     o%y1 = y1
     o%x2 = x2
     o%y2 = y2
     return
    END SUBROUTINE new_box

  END MODULE box_class
```

Biblioteka graficzna DISLIN

Helmut Michels pisze o swojej bibliotece graficznej: (…) The data plotting library DISLIN IS written in the programming languages Fortran and C. The name DISLIN is an abbreviation for Device-Independent Software LINdau since applications were designed to run on different computer systems without any changes. The library contains subroutines and functions for displaying data graphically as curves, bar graphs, pie charts, 3-D colour plots, surfaces, contours and maps. DISLIN supports now several hardware platforms, operating systems and compilers. A real Fortran 90 library is available for most Fortran90 compilers (…)

DISLIN is free for the operating systems Linux and FreeBSD and for the GNU compilers G95, GCC+G77/MS-DOS and GCC+G77/Windows 9x/NT. Other DISLIN versions are available at low charge and can be tested free of charge. Programs linked with DISLIN can be distributed without any royalties together with neccessary shareable DISLIN libraries.

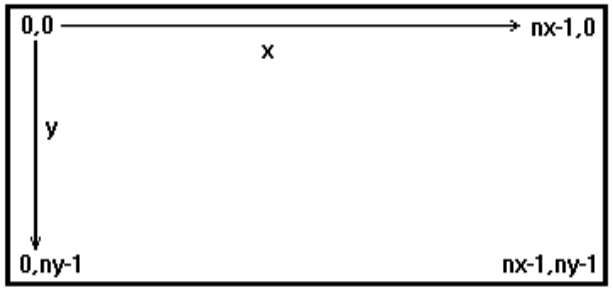
Wykład przygotowano w oparciu o darmową dystrybucję biblioteki DISLIN
dla kompilatora fortranu GNU G95 (GNU Licence).
Darmowa, biblioteka DISLINdo do pobrania z http://www.dislin.de.
Darmowy kompilator fortranu GNU G95, do pobrania z: http://g95.sourceforge.net/.

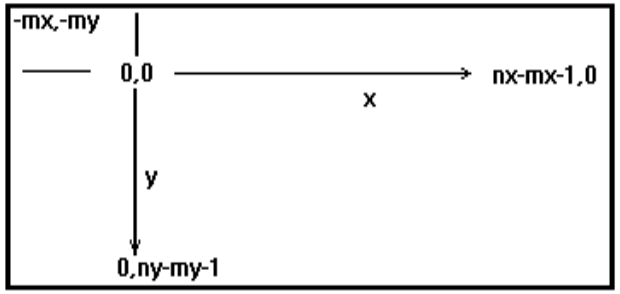
Autorem biblioteki DISLIN jest:

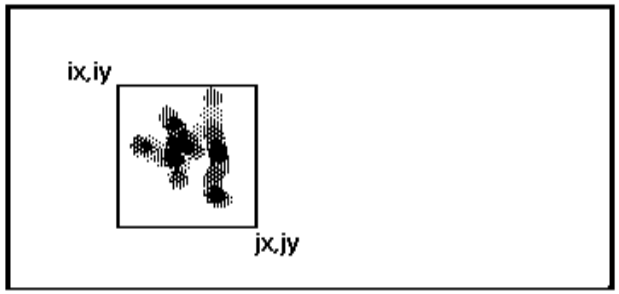
Helmut Michels
Max-Planck-Institut fuer Aeronomie
D-37191 Katlenburg-Lindau, Max-Planck-Str. 2, Germany
E-Mail: michels@linmpi.mpg.de
Tel.: +49 5556 979 334, Fax: +49 5556 979 240

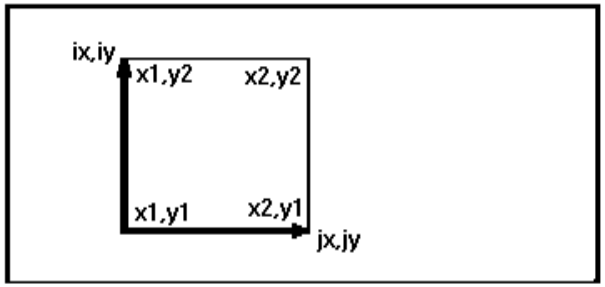
Współrzędne graficzne

1. współrzędne fizyczne;
2. współrzędne strony/portu/ (obszar ekranu, gdzie jest rysowana grafika);
5. współrzędne układu współrzędnych użytkownika.









Obsługiwane typy terminali graficznych: VGA, X Windows, Windows API, i Tektronix.

Podstawowe założenia:

Obszar tworzenia grafiki ograniczony jest do prostokatnego obszaru zwanego STRONĄ.

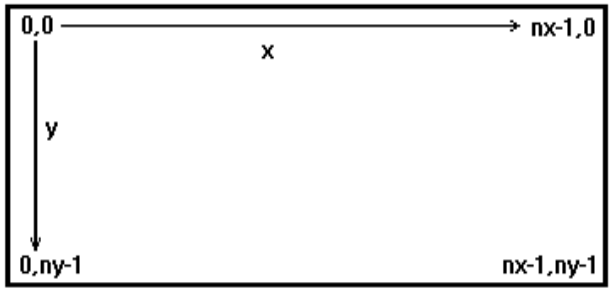
Wszelkie obiekty graficzne które miałyby być kreślone poza granicą strony są obcinane.

Każdy segment programu wywołujący procedury biblioteki DISLIN musi mieć przyłączony MODUŁ DISLIN.

Np.
```
PROGRAM TEST
  USE DISLIN
  CALL DISINI ()
  CALL MESSAG ('This is a test', 100, 100)
  CALL DISFIN ()
END PROGRAM TEST
```

W momencie uruchomienia biblioteki DISLIN przyjmowana jest przez domniemanie strona o parametrach:

Rodzielczośc 100 px/cm, (0,0) w lewym górnym rogu, rozmiar DIN A4 “landscape”



(2969,2099) dla DIN A4

Formaty graficzne: GKSLIN, CGM, HPGL, PostScript, PDF, Prescribe, WMF, PNG, PPM, BMP, TIFF.

DISLIN może tworzyć nastepujące formaty obrazów:

1. PostScript
2. EPS
3. PDF
4. HPGL
5. WMF
6. Java applet file (vectors and colours are stored in form of a Java applet).
7. SVG file (Scalable Vector Graphics (SVG) is a language for describing graphics in XML)
8. TIFF
9. PNG
10. PPM
11. BMP
12. IMAGE
13. Tektronix, VGA and X Window

STRUKTURA POZIOMÓW BIBLIOTEKI DISLIN

Większość procedur DISLIN'a może byc wywoływane w dowolnym momencie działania programu. Niektóre procedury **muszą być jednak wywołane w ściśle zdefiniowanej kolejności**. DISL:IN używa pojecia POZIOMÓW dla kontroli poprawnosci kolejnosci wywołania procedur. Ustalono nastepujące poziomy:

0 przed inicjalizacją działania biblioteki lub po zakończeniu

1 po inicjalizacji lub wywołania procedury ENDGRF

2 po wywołaniu procedury GRAF

3 po wywołaniu procedur GRAF3 lub GRAF3D.

OGÓLNIE, FRAGMENT SEGMENTU UŻYWAJĄCY GRAFIKI POWINIEN MIEĆ NASTĘPUJĄCĄ STRUKTURĘ:

(1) ustalenie formatu strony lub formatu zbioru i jego nazwy

(2) inicjalizacja biblioteki graficznej

(3) ustawienie parametrów rysowania

(4) rysowanie układu współrzędnych

(5) wypisanie tytułów, etc.

(6) rysowanie danych

(7) zakończenie.

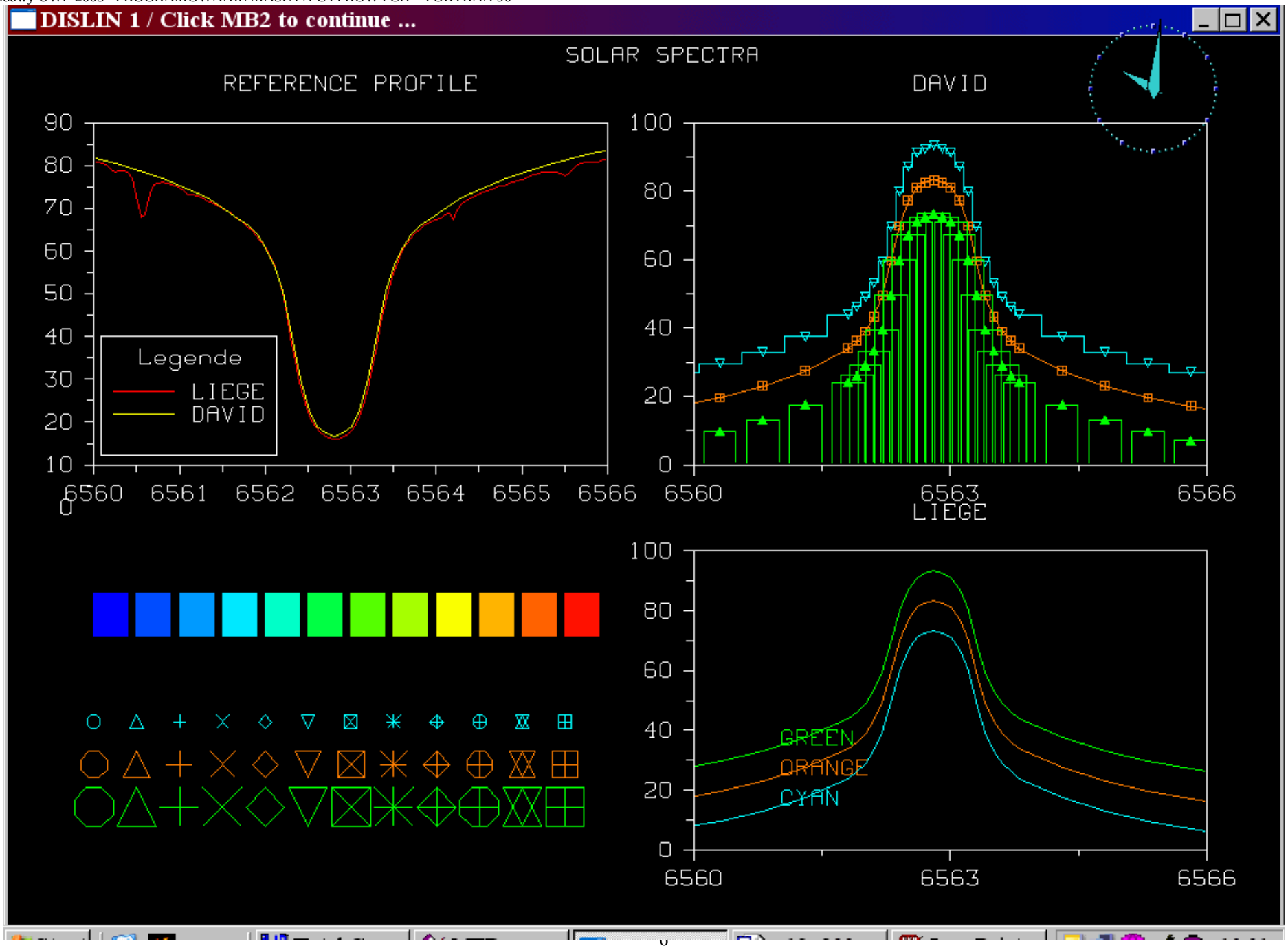
Uwagi: numery urządzeń logicznych 15, 16 i 17 są zarezerwowane przez DISLIN dla celów rysowania i dla zbiorów parametrów.

ERROR MESSAGES

When a DISLIN subroutine or function is called with an illegal parameter or not according to the level structure, DISLIN writes a warning to the screen. The call of the routine will be ignored and program execution resumed. Points lying outside of the axis system will also be listed on the screen. Error messages can be suppressed or written to a file with the routines ERRMOD and ERRDEV.

UTILITY PROGRAMS

D I S H L P – wyswietla okienkową wersję listy procedur i ich opisów.
D I S M A N – wyswietla okienkową wersję podręcznika biblioteki DISLIN
D I S D R V – wysyła zbiór do urządzenia drukującego.
D I S I M G – wyświetla obraz (*.img) na ekranie lub zamienia go ns PS lub TIFF
D I S M O V – wyświetla serie obrazów na ekranie (utworzonych proc. RIMAGE)
D I S T I F – …
D I S A P S - …


DISLIN 1 / Click MB2 to continue ...
SOLAR SPECTRA
REFERENCE PROFILE
DAVID
Legende
LIEGE
DAVID
LIEGE
GREEN
ORANGE
CYAN

Będziemy pracować z następującym programem:

```
PROGRAM FOO

USE DISLIN

    IMPLICIT NONE
    INTEGER :: i,k,nmin, KK,IM, NN,JJ
    real :: rminpg,rminlieg
    real :: liege(121),liegelam(121),dav(33),davlam(33),X(33),Y(33),dv(17),dl(17)
    CHARACTER(LEN=6) :: CPOL(3)=(/'STEP  ','LINEAR','BARS  '/)
    CHARACTER(LEN=6) :: CKOL(3)=(/'CYAN  ','ORANGE','GREEN '/)
    CHARACTER(LEN=6) :: CTYP(3)=(/'SOLID ','DOT   ','DASHL '/)
    character(len=100) :: lab

    dv=(/16.6,17.5,18.9,22.8,30.2,40.4,50.6,56.8,61.0,63.8,66.0,&
         72.5,77.0,80.3,82.9,84.9,86.6/)
    dl=(/0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.5,2.0,2.5,&
         3.0,3.5,4.0/)
    liege=(/81.047,80.692,80.384,79.867,79.008,78.512,78.642,&
         78.651,78.229,76.979,72.907,67.885,68.470,73.610,&
         75.457,75.973,76.058,75.771,75.495,75.122,74.732,&
         74.166,73.199,73.113,73.024,72.565,72.036,71.561,&
         71.062,70.521,69.925,69.339,68.683,67.984,67.229,&
         66.425,65.604,64.624,63.577,62.435,60.989,59.346,&
         57.003,54.040,50.189,45.374,39.825,34.263,29.320,&
         25.340,22.443,20.264,18.675,17.558,16.763,16.254,&
         16.070,16.083,16.378,16.969,17.883,19.276,21.197,&
         23.879,27.648,32.443,37.872,42.914,47.129,51.556,&
         54.832,57.518,59.862,61.545,62.847,63.983,65.013,&
         65.844,66.568,67.056,67.422,67.514,68.420,68.771,&
         67.417,69.535,71.124,71.856,72.316,72.846,73.253,&
         73.701,74.068,74.469,74.854,75.148,75.355,75.780,&
         76.159,76.487,76.789,77.119,77.486,77.749,78.037,&
         78.339,78.568,78.460,78.409,78.268,77.570,78.053,&
         79.282,80.145,80.498,80.730,80.864,80.915,80.911,&
         81.338,81.617/)

    do i=1,121
      liegelam(i)=6560.0+(i-1)*0.05
    end do
    rminlieg=10000.
    do k=1,121
      if (liege(k).lt.rminlieg) rminlieg=liege(k)
    end do

    do i=1,17
     dav(16+i)=dv(i)
     davlam(16+i)=dl(i)
    end do
    do i=1,16
     dav(i)=dv(18-i)
     davlam(i)=-1.0*dl(18-i)
    end do
    do i=1,33
      davlam(i)=davlam(i)+6562.808
    end do
```

```
      CALL METAFL('cons')
      CALL SETPAG('DA4L')
      CALL DISINI ()
      call errmod('ALL','OFF')
      CALL MESSAG('SOLAR SPECTRA',1300,25)

      CALL AXSPOS(200,1000)
      CALL AXSLEN(1200,800)
      CALL NAME('Lambda','X')
      CALL NAME('Intensity','Y')
      CALL LABDIG(-1,'XY')
      CALL TICKS(-1,'XY')
      CALL TITLIN('REFERENCE PROFILE',4)
      CALL SETGRF('LABELS','LABELS','LINE','LINE')
      CALL GRAF(6560.0,6566.0,6560.0,1.0,10.0,90.0,0.0,10.0)
      CALL TITLE ()
      call legini(lab,3,6)
      CALL COLOR('RED')
      CALL CURVE(LIEGELAM,LIEGE,121)
      call leglin(lab,'LIEGE',1)
      CALL COLOR('YELLOW')
      CALL CURVE(DAVLAM,DAV,33)
      call leglin(lab,'DAVID',2)
      CALL COLOR('WHITE')
      call legend(lab,5)
      call endgrf ()


      CALL AXSPOS(1600,1000)
      CALL AXSLEN(1200,800)
      CALL TITLIN('DAVID',4)
      CALL SETGRF('LABELS','LABELS','LINE','LINE')
      CALL GRAF(6560.0,6566.0, 6560.0,3.0, 0.0,100.0, 0.0,20.0)
      CALL TITLE ()
      CALL COLOR('YELLOW')
      CALL INCMRK(1)
      CALL HSYMBL(20)
      do NN=1,3
            CALL MARKER(NN*6)
            CALL COLOR(CKOL(NN))
            CALL POLCRV(CPOL(NN))
            CALL CURVE(DAVLAM,120.-(10.0*NN+DAV),33)
      end do
      CALL COLOR('WHITE')
      call endgrf ()

      CALL AXSPOS(1600,1900)
      CALL AXSLEN(1200,700)
      CALL TITLIN('LIEGE',4)
      CALL SETGRF('LABELS','LABELS','LINE','LINE')
      CALL GRAF(6560.0,6566.0, 6560.0,3.0, 0.0,100.0, 0.0,20.0)
      CALL TITLE ()
      CALL POLCRV('LINEAR')
      CALL INCMRK(0)
      do NN=1,3
            CALL COLOR(CKOL(4-NN))
            CALL MESSAG (ckol(4-
nn),NXPOSN(6561.0),NYPOSN(50.0-NN*10.))
            CALL CURVE(DAVLAM,120.-(10.0*NN+DAV),33)
      end do
      call endgrf ()

      do nn=1,12
        CALL RECFLL(100+100*nn,1300,80,100,nn*20+10)
      end do

      do i=1,3
      CALL HSYMBL(30*i)
      do nn=1,12
        call color(ckol(i))
        call symbol(nn,100+100*nn,1500+100*i)
      end do
      end do

     CALL DISFIN ()

stop
END program foo
```

METAFL

METAFL defines the metafile format.

CALL METAFL (CFMT) level 0

CFMT - is a character string that defines the file format.
'GKSL', 'CGM', 'PS' - coloured PostScript file, 'EPS', 'POST' - greyscaled PostScript file.
'PSCL' - coloured PostScript file with a black bckgrd, 'PDF', 'KYOC',
'HPGL', 'JAVA','WMF','SVG','TIFF','PNG','PPM','IMAG' - defines an image file,
'BMP', 'VIRT' defines a virtual file. The metafile is hold in a raster format in computer memory.
'CONS' defines a graphics output on the screen. If the screen is a windows display, a graphical window is used that has nearly the size of the screen.
'XWIN' defines an X Window display. By default, the size of the window is nearly 2/3 of the size of the screen.
Default: CFMT = 'GKSL'.

SETPAG

SETPAG selects a predefined page format.

CALL SETPAG (CPAGE) level 0

CPAGE is a character string that defines the page format.
np.

= 'DA4L' DIN A4, landscape, 2970 * 2100 Punkte.
= 'DA4P' DIN A4, portrait, 2100 * 2970 Punkte. Default: CPAGE = 'DA4L'.

DISINI

DISINI initializes DISLIN by setting default parameters and creating a plotfile. **The level is set to 1**. DISINI must be called before any other DISLIN routine except for those noted throughout the manual.

CALL DISINI ( ) level 0

DISFIN

DISFIN terminates DISLIN and prints a message on the screen. The level is set back to 0.

CALL DISFIN ( ) level 1, 2, 3

ERRMOD

The printing of warnings and the output of the protocol in DISFIN can be disabled with the routine ERRMOD.

CALL ERRMOD (CKEY, CMOD) level 0

CKEY is a character string that can have the values 'WARNINGS', 'CHECK', 'PROTOCOL' and 'ALL'.
'WARNINGS' means the error messages about bad parameters passed to DISLIN routines,
'CHECK' the out of range check of coordinates passed to plotting routines such as CURVE
'PROTOCOL' the output of the protocol in DISFIN.

CMOD is a character string that can have the values 'ON' and 'OFF'.

Default: **('ALL', 'ON')**

MESSAG

MESSAG plots text.

CALL MESSAG (CSTR, NX, NY) level 1, 2, 3

CSTR łańcuch o dług. <= 256 znaków
NX, NY współrzędne lewego-górnego narożnika napisu (we współrzednych strony).

Np.: **CALL MESSAGE ('zaczynamy…',200,300)**

AXSPOS

AXSPOS determines the position of an axis system.

CALL AXSPOS (NXA, NYA) level 1

NXA, NYA are plot coordinates that define the lower left corner of an axis system. By default, axis systems are centred in the X-direction while NYA is set to the value (page height-300).

AXSLEN

AXSLEN defines the size of an axis system.

CALL AXSLEN (NXL, NYL) level 1

NXL, NYL are the length and height of an axis system in plot coordinates. The default values are set

to 2/3 of the page length and height.

SCALE

This routine sets the axis scaling to logarithmic or linear.

CALL SCALE (CSCL, CAX) level 1, 2, 3

CSCL = 'LIN' denotes linear scaling. = 'LOG' denotes logarithmic scaling.
CAX is a character string that defines the axes.
efault: ('LIN', 'XYZ').
Note: For logarithmic scaling, the corresponding parameters in GRAF must be exponents of base 10.

SETGRF

SETGRF removes a part of an axis or a complete axis from an axis system.

CALL SETGRF (C1, C2, C3, C4) level 1, 2, 3

Ci are character strings corresponding to the four axes of an axis system. C1 corresponds to the lower X-axis, C2 to the left Y-axis, C3 to the upper X-axis and C4 to the right Y-axis. The parameters can have the values 'NONE', 'LINE', 'TICKS', 'LABELS' and 'NAME'.
'NONE', complete axes will be suppressed,
'LINE', only axis lines will be plotted,
'TICKS', axis lines and ticks will be plotted,
'LABELS', axis lines, ticks and labels will be plotted
'NAME', all axis elements will be displayed.
Default: ('NAME', 'NAME', 'TICKS', 'TICKS').

Notes: - By default, GRAF plots a frame of thickness 1 around axis systems. Therefore, in addition to the parameter 'NONE', FRAME should be called with the parameter 0 for suppressing complete axes.

NAME

NAME defines axis titles.

CALL NAME (CSTR, CAX) level 1, 2, 3

CSTR is a character string containing the axis title (<= 132 characters).
CAX is a character string that defines the axes. Default: (' ', 'XYZ').

LABDIG

The routine LABDIG defines the number of decimal places in the labels.

CALL LABDIG (NDIG, 'BARS') level 1, 2, 3

NDIG is the number of decimal places (>= -1). Default: NDIG = 1

LABPOS

LABPOS defines the position of labels.

CALL LABPOS (CPOS, CAX) level 1, 2, 3

CPOS is a character string defining the position.

= 'TICKS' means that labels will be plotted at major ticks.

= 'CENTER' means that labels will be centred between major ticks.
= 'SHIFT' means that the starting and end labels will be shifted.

CAX is a character string that defines the axes. Default: ('TICKS', 'XYZ').

LABELS

LABELS determines which label types will be plotted on an axis.

CALL LABELS (CLAB, CAX) level 1, 2, 3

CLAB is a character string that defines the labels.

= 'NONE' will suppress all axis labels.
= 'FLOAT' will plot labels in floating-point format.
= 'EXP' will plot floating-point labels in exponential format where fractions range between 1 and 10.
= 'FEXP' will plot labels in the format fEn where f ranges between 1 and 10.
= 'LOG' will plot logarithmic labels with base 10 and the corresponding exponents.
= 'CLOG' is similar to 'LOG' except that the entire label is centred below the tick mark; with 'LOG', only the base '10' is centred.
= 'ELOG' will plot only the logarithmic values of labels.
= 'TIME' will plot time labels in the format 'hhmm'.
= 'HOURS' will plot time labels in the format 'hh'.
= 'SECONDS' will plot time labels in the format 'hhmmss'.
= 'DATE' defines date labels.

CAX is a character string that defines the axes.
Default: ('FLOAT', 'XYZ').

RGTLAB

The routine RGTLAB right-justifies user labels. By default, user labels are left-justified.

CALL RGTLAB () level 1, 2, 3

TICKS

This routine is used to define the number of ticks between axis labels.

CALL TICKS (NTIC, CAX) level 1, 2, 3

NTIC is the number of ticks (>= 0).

CAX is a character string that defines the axes.

Default: (2, 'XYZ').

TITLIN

This subroutine defines up to four lines of text used for axis system titles. The text can be plotted with TITLE after a call to GRAF.

CALL TITLIN (CSTR, IZ) level 1, 2, 3

CSTR is a character string (<= 132 characters).
IZ is an integer that contains a value between 1 and 4 or -1 and -4. If IZ is negative, the

line will be underscored. Default: All lines are filled with blanks.

TITPOS

The routine TITPOS defines the position of title lines which can be plotted above or below axis systems.

CALL TITPOS (CPOS) level 1, 2, 3

CPOS is a character string that can have the values 'ABOVE' and 'BELOW'.
Default: CPOS = 'ABOVE'.

UKŁAD WSPÓŁRZĘDNYCH

An axis system defines an area on the page for plotting data. Various axis systems can be plotted to accommodate different applications. For two-dimensional graphics, a maximum of two parallel X- and Y-axes can be drawn. The axis system is scaled to fit the range of data points and can be labeled with values, names and ticks. Two-dimensional axis systems are plotted with a call to the routines GRAF or POLAR.

GRAF

GRAF plots a two-dimensional axis system.

CALL GRAF (XA, XE, XOR, XSTEP, YA, YE, YOR, YSTEP) level 1

XA, XE min i max wartosci dla osi X
XOR, XSTEP położenie pierwszej labelki X i krok labelek
YA, YE min i max wartosci dla osi Y
YOR, YSTEP położenie pierwszej labelki X i krok labelek

Notes: - GRAF must be called in level 1 and **automatically sets the level to 2**.
When plotting more than 1 axis system on a page, **ENDGRF must be called** in between each new set of axes in order to set the level back to 1.

CALL GRAF (6560.0,6566.0,6560.0,1.0,0.0,100.0,0.0,10.0)

TITLE

TITLE plots a title over an axis system. The title may contain up to four lines of text designated with TITLIN.

CALL TITLE () level 2, 3

HTITLE

HTITLE defines the character height for titles. The character height defined by HEIGHT will be used if HTITLE is not called.

CALL HTITLE (NHCHAR) level 1, 2, 3

NHCHAR is the character height in plot coordinates.

CURVE

CURVE connects data points with lines or plots them with symbols.

CALL CURVE (XRAY, YRAY, N) level 2, 3

XRAY, YRAY tablice zawierające współrzędne X- i Y- kolejnych punktów.
For a polar scaling, XRAY must hold the radial values and YRAY the angular values expressed in radians.
N ilośc punktów.

Notes: - CURVE must be called after GRAF from level 2 or 3.

- By default, data points that lie outside of an axis system are listed on the screen. The listing can be suppressed with the routine NOCHEK.
- CURVE suppresses lines outside the borders of an axis system. Suppressing can be disabled with NOCLIP or the margins of suppression can be changed with GRACE.
- INCMRK determines if CURVE plots lines or symbols.
- When plotting several curves, attributes such as colour and line style can be changed automatically by DISLIN or directly by the user.
  The routine CHNCRV defines which attributes are changed automatically.
- Different data interpolation methods can be chosen with POLCRV.

LEGINI

LEGINI initializes a legend.

CALL LEGINI (CBUF, NLIN, NMAX) level 1, 2, 3

CBUF is a character variable used to store the lines of text in the legend. The variable must be defined by the user to have at least NLIN*NMAX characters.
NLIN is the number of text lines in the legend.
NMAX is the number of characters in the longest line of text.

LEGLIN

LEGLIN stores lines of text for the legend.

CALL LEGLIN (CBUF, CSTR, ILIN) level 1, 2, 3

CBUF see LEGINI.
CSTR is a character string that contains a line of text for the legend.
ILIN is the number of the legend line between 1 and NLIN.

LEGEND

LEGEND plots legends.

CALL LEGEND (CBUF, NCOR) level 2, 3

CBUF see LEGINI.[0.3cm]
NCOR indicates the position of the legend (1-8...)

- LEGTIT (CTIT) sets the title of the legend.
- LEGPOS (NX, NY) defines a global position for the legend where NX and NY are the plot coordinates of the upper left corner. After a call to LEGPOS, the second parameter in LEGEND will be ignored.
- NLX = NXLEGN (CBUF) and NYL = NYLEGN (CBUF)
  return the length and the height of a legend in plot coordinates.
- FRAME (NFRA) defines the thickness of a frame plotted around a legend.
- LINESP (XF) changes the spacing of lines in a legend.

- LEGCLR retains the same colour for curves and lines of text in the legend.

COLOR

COLOR defines the colours used for plotting text and lines.

CALL COLOR (CNAME) level 1, 2, 3

CNAME is a character string that can have the values 'BLACK', 'RED', 'GREEN', 'BLUE', 'CYAN', 'YELLOW', 'ORANGE', 'MAGENTA', 'WHITE', 'FORE' and 'BACK'. The keyword 'FORE' resets the colour to the default value, while the keyword 'BACK' sets the colour to the background colour.

Note: Colours can also be defined with SETCLR which selects a colour index from an actual colour table (see chapter 11).

INCMRK

INCMRK selects line or symbol mode for CURVE.

CALL INCMRK (NMRK) level 1, 2, 3

NMRK = - n means that CURVE plots only symbols. Every n-th point will be marked by a symbol.
= 0 means that CURVE connects points with lines.
= n means that CURVE plots lines and marks every n-th point with a symbol.
Default: NMRK = 0

MARKER

The symbols used to plot points can be selected with the routine MARKER. The symbol number will be incremented by 1 after a certain number of calls to CURVE defined by INCCRV.

CALL MARKER (NSYM) level 1, 2, 3

NSYM is the symbol number between 0 and 21. The symbols are shown in appendix B.
Default: NSYM = 0

POLCRV

The routine POLCRV defines an interpolation method used by CURVE to connect points.

CALL POLCRV (CPOL) level 1, 2, 3

CPOL is a character string containing the interpolation method.
= 'LINEAR' defines linear interpolation.
= 'STEP' defines step interpolation.
= 'STAIRS' defines step interpolation.
= 'BARS' defines bar interpolation.
= 'FBARS' defines filled bar interpolation.
= 'STEM' defines stem interpolation.
= 'SPLINE' defines spline interpolation.
= 'PSPLINE' defines parametric spline interpolation.

Default: CPOL = 'LINEAR'.

RECFLL

The routine RECFLL plots a coloured rectangle where the position is determined by the upper left corner.

CALL RECFLL (NX, NY, NB, NH, NCOL) level 1, 2, 3

NX, NY are the plot coordinates of the upper left corner.
NB, NH are the width and height in plot coordinates.
NCOL is a colour index in the range 0 to 255.

POINT

The routine POINT plots a coloured rectangle where the position is determined by the centre.

CALL POINT (NX, NY, NB, NH, NCOL) level 1, 2, 3

NX, NY are the plot coordinates of the centre point.
NB, NH are the width and height in plot coordinates.
NCOL is a colour index in the range 0 to 255.

RLPOIN

The routine RLPOIN plots a coloured rectangle where the position is specified in user coordinates.

CALL RLPOIN (X, Y, NB, NH, NCOL) level 2, 3

Note: RLPOIN clips rectangles at the borders of an axis system.

SYMBOL

The routine SYMBOL plots symbols.

CALL SYMBOL (NSYM, NX, NY) level 1, 2, 3

NSYM is a symbol number between 0 and 23.
NX, NY is the centre of the symbols in plot coordinates.

RLSYMB

RLSYMB plots a symbol where the centre is specified by user coordinates.

CALL RLSYMB (NSYM, XP, YP) level 2, 3

NSYM is a symbol number between 0 and 21.
XP, XP is the centre of the symbol in user coordinates.