Materiały do ćwiczeń z programowania w Fortranie 90

Andrzej Eilmes
Zakład Metod Obliczeniowych Chemii UJ
Kraków 2007

0. Wyjaśnienia

0.1. Czym ten tekst nie jest?

Poniższy tekst nie jest podręcznikiem programowania w Fortranie, czy innym języku, ani podręcznikiem algorytmów, czy metod numerycznych.

0.2. Czym ten tekst jest?

Jest on próbą skrótowego zebrania podstawowych informacji potrzebnych studentowi chemii do zaliczenia kursu programowania w jęz. Fortran. Omawia on standard języka znany jako Fortran 90, bazując na opisie podanym w:

J.C. Adams, W. S. Brainerd, J.T. Martin, B.T. Smith, J. L. Wagner, *Fortran 90 Handbook. Complete ANSI/ISO Reference.*

Ponieważ jest to bardzo uproszczony opis języka, stosowana terminologia także jest uproszczona.

0.3. Notacja

Kod programu w Fortranie oraz pojedyncze polecenia będące częścią języka zapisywane są czcionką o stałej szerokości. Pochylony krój pisma sygnalizuje ogólną nazwę lub ogólny zapis wyrażenia.

1. Pierwsze kroki

1.1. Jak zapisać i uruchomić program w Fortranie?

Do zapisu programu w języku Fortran używamy następujących symboli: liter alfabetu łacińskiego (a-z), cyfr (0-9), odstępu (spacji) oraz znaków

= + - * / ( ) , . ‘ (apostrof) _ (podkreślenie) : ! ” % & ; < >

Inne znaki mogą pojawić się w stałych tekstowych. Z wyjątkiem stałych tekstowych duże i małe litery są utożsamiane.

Poprawny identyfikator obiektu (nazwa) w Fortranie 90 jest ciągiem maksymalnie 31 znaków składającym się z liter, cyfr i znaku podkreślenia, zaczynającym się od litery. Duże i małe litery są utożsamiane, zatem wszystkie poniższe nazwy:

```
Alamakota
alamaKoTA
alamakota
```

są równoważne.

W obrębie jednostki leksykalnej (ang. *token*), czyli najmniejszej rozpoznawalnej porcji znaków (nazwa zmiennej, słowo kluczowe, zapis stałej lub operatora) nie mogą pojawić się odstępy. Są one dopuszczalne między jednostkami i wtedy ich liczba nie ma znaczenia – tzn. dwa lub więcej odstępów jest traktowane jako jeden, np. zapisy:

```
deltax=x1-x0
deltax = x1 - x0
deltax=     x1 - x0
```

są równoważne, natomiast zapis:

```
delta x = x1-x0
```

jest błędny, bo spacja pojawiła się wewnątrz nazwy.

Kod źródłowy programu w Fortranie możemy zapisać w dowolnym edytorze tekstowym, pod warunkiem, że nie dodaje on do zapisywanego tekstu dodatkowych znaków. Wygodnie jest używać jednego z programów rozpoznających składnię języka, np. `emacs`, `kwrite` czy `nedit`.
Plik z zapisanym kodem źródłowym powinien mieć rozszerzenie wskazujące na jego zawartość, czyli .f90, np. `okaz.f90`. W naszym przypadku pliki powinny mieć rozszerzenie f90.

Aby napisać kod źródłowy programu, uruchamiamy odpowiedni edytor, np. wydając w linii poleceń rozkaz

```
kwrite okaz.f90 &
```

Parametrem tego polecenia jest nazwa edytowanego pliku (w naszym przypadku `okaz.f90`).
Dzięki dodaniu symbolu & na końcu będziemy mogli wydawać w linii poleceń dalsze rozkazy (kompilacja) bez zamykania okna edytora. Kod źródłowy należy zapisać na dysku.

Przed uruchomieniem programu niezbędna jest jego kompilacja, czyli przetłumaczenie na zestaw instrukcji procesora danej maszyny (kod wykonywalny).
Pod systemem operacyjnym Debian zainstalowanym w pracowni studenckiej kompilator Fortranu 90/95 nazywa się `gfortran`, parametrem jego wywołania jest nazwa (nazwy) plików, które należy przetłumaczyć.
Aby skompilować program `okaz.f90`, możemy zatem wydać w linii poleceń terminala polecenie

```
gfortran okaz.f90
```

Jeśli kompilacja przebiegnie bezbłędnie, na dysku zostanie zapisany kod wykonywalny pod nazwą `a.out` . Możemy go uruchomić wydając polecenie

```
./a.out
```

Aby utworzyć plik wykonywalny o określonej nazwie, polecenie `gfortran` uzupełniamy o parametr `-o` , po którym podajemy żądaną nazwę pliku wykonywalnego:

`gfortran -o` *nazwa_pliku_wykonywalnego nazwa_pliku_źródłowego*

W naszym przypadku wyglądać może to następująco:

```
gfortran -o okaz.x okaz.f90
```

W wyniku zadziałania tego polecenia program zawarty w pliku `okaz.f90` zostanie skompilowany a kod wykonywalny zapisany w pliku `okaz.x` (rozszerzenie tej nazwy może być dowolne lub może go nie być wcale; tu użyliśmy rozszerzenia .x dla zaznaczenia, że chodzi o plik wykonywalny (eXecutable)). Skompilowany w ten sposób program uruchomimy poleceniem:

```
./okaz.x
```

1.2. Jak pisać kod programu w Fortranie 90/95?

Kod źródłowy programu w Fortranie składa się z instrukcji i specyfikacji (dyrektyw). Instrukcje kodują polecenia wykonywane podczas działania programu, natomiast specyfikacje zawierają informacje dotyczące interpretacji instrukcji wykorzystywane przez kompilator podczas tłumaczenia kodu źródłowego.

Program fortranowski składa się z oddzielnych jednostek (ang. *units*). Każdy program fortranowski musi mieć dokładnie jeden tzw. program główny – od niego rozpoczyna się działanie programu. Dodatkowe jednostki mogą (lecz nie muszą) wystąpić w miarę potrzeby. Na razie zajmiemy się tylko programami składającymi się wyłącznie z programu głównego.

Specyfikacje obowiązujące w danej jednostce muszą znaleźć się na jej początku przed pierwszą instrukcją wykonywalną.

Poniżej opiszemy zasady jednego z dwu sposobów zapisywania kodu źródłowego, tzw. *free form source*.

Polecenia zapisujemy w kolejnych liniach pliku. Linia może mieć do 132 znaków. W jednym wierszu kodu można zapisać kilka poleceń oddzielając je znakiem `;` , np.

```
y=0;x=2;z=x+y
```

Wykrzyknik (`!`) służy do zaznaczenia komentarza (z wyjątkiem wystąpienia w obrębie stałej tekstowej) – tekst zapisany w wierszu na prawo od wykrzyknika nie wpływa na działanie programu (nie jest interpretowany przez kompilator).

W przypadku długich poleceń może okazać się potrzebne zapisanie ich w dwu lub więcej wierszach. Wykorzystujemy w tym celu `&` jako znak kontynuacji. Umieszczony na końcu wiersza oznacza, że zapis w wierszu następnym jest kontynuacją poprzedniego. Przyjmijmy zdroworozsądkową zasadę, że przy tworzeniu linii kontynuacji nie rozdziela się jednostki leksykalnej między dwa wiersze. Program zyska dzięki temu na przejrzystości a my nie będziemy musieli zastanawiać się nad pułapkami, jakie mogłyby pojawić się w przeciwnym przypadku.

1.3. Zapiszmy najprostszy program fortranowski.

Program główny musi kończyć się instrukcją `end`. Instrukcja ta kończy pracę całego programu a jednocześnie podczas kompilacji stanowi dla kompilatora dyrektywę zaznaczającą koniec tłumaczonej jednostki.

Na początku programu głównego może (lecz nie musi) pojawić się dyrektywa

`program` *nazwa*

gdzie *nazwa* to poprawna fortranowska nazwa nadana danemu programowi (patrz 1.1). Nie musi mieć ona nic wspólnego (choć dla wygody powinna) z nazwą pliku, w którym został zapisany kod programu.

Może zatem wyglądać następująco:

```
program okaz          ! dobrze
```

Poniższa instrukcja jest nieprawidłowa:

```
program okaz.f90      ! zle
```

ponieważ w nazwie nie może wystąpić kropka.

Dyrektywa ta może być pominięta (kompilator i bez niej jest w stanie rozpoznać początek programu głównego)

Możemy zatem zapisać kod prostego programu fortranowskiego:

```
program okaz
end
```

Instrukcja `end` kończąca program ma ogólną postać

`end program` *nazwa*

przy czym słowo `program` oraz nazwę można pominąć, jak w przykładzie powyżej.

Ponieważ dyrektywę `program` także można pominąć, ostatecznie otrzymujemy najkrótszy poprawny program w Fortranie:

```
end
```

Program ten jest bezbłędny, tym niemniej po uruchomieniu natychmiast skończy pracę nie dając żadnych efektów swojej działalności.
Skłońmy go zatem do wypisania jakiegoś komunikatu dodając instrukcję `write`:

```
program okaz
write(*,*) 'Witamy !'
end program okaz
```

Instrukcję `write` omówimy szczegółowo później, na tym etapie wystarczy wiedzieć, że:
- gwiazdki w nawiasach oznaczają wypisanie tekstu na standardowe wyjście (ekran) w sposób niezredagowany (bez określenia, jak wydruk ma wyglądać)
- tekst w apostrofach to stała tekstowa, która będzie wypisana przez instrukcję `write`; zwróćmy uwagę na to, że wykrzyknik w tym przypadku nie oznacza kontynuacji, ponieważ występuje w obrębie stałej tekstowej

W wyniku zadziałania powyższego programu powinniśmy zobaczyć na monitorze tekst
Witamy !

Zadanie:

1. „Zepsuj" kod powyższego programu robiąc jakiś błąd (przekręcając któreś ze słów, wstawiając złą nazwę programu, usuwając któryś nawias, apostrof lub gwiazdkę, usuwając instrukcję `end`). Sprawdź, jaka będzie reakcja kompilatora.

2. Aby zorientować się dlaczego najprawdopodobniej nie chcesz wiedzieć, jakie programy uruchamia polecenie `gfortran` podczas kompilacji, wywołaj je dodając dodatkowo opcję `-v`, np.

```
gfortran -v -o okaz.x okaz.f90
```

2. Jak obliczyć wartość prostego wyrażenia?

2.1. Typy

W Fortranie stałe i zmienne są określonego typu – określającego jak reprezentowane są w pamięci maszyny. Wśród typów podstawowych języka na tym etapie interesować nas będą typy numeryczne: `integer` (całkowity), `real` (rzeczywisty) i `complex` (zespolony).

W przypadku typów numerycznych możemy dodatkowo określić, jaki ma być zakres przechowywanych przez nie wartości i/lub liczba miejsc znaczących. W szczególności standard Fortranu 90 wymaga, aby dostępne były przynajmniej dwa rodzaje typu rzeczywistego. Na razie nie będziemy się zajmować tym problemem.

2.2. Stałe

Zapisem stałej całkowitej jest ciąg cyfr dziesiątkowych, poprzedzony znakiem (znak + może być pominięty), bez separatora dziesiętnego, np.

```
+2
-10
0
23
2000000000
```

Stałą rzeczywistą możemy zapisać jako liczbę rzeczywistą z kropką jako separatorem dziesiętnym, lub liczbę rzeczywistą z wykładnikiem. Zerową część całkowitą albo zerową część ułamkową (ale nie obie jednocześnie) można pominąć. Częścią wykładniczą stałej rzeczywistej jest stała całkowita (ze znakiem lub bez) podana po literze `E`.

Przykładowe zapisy stałych rzeczywistych:

```
2.0
-1.0
0.0
1.
.0001
1e-10   (czyli 1 × 10^-10)
2E03    (czyli 2 × 10^3)
2.e3    (jak wyżej)
-1.12e-101      (czyli -1.12 × 10^-101)
4E-8        (czyli 4 × 10^-8)
1e0         (czyli 1 × 10^0)
```

Należy pamiętać, że w Fortranie typ odnosi się do sposobu reprezentacji danych w pamięci a nie do przynależności do odpowiedniego zbioru w sensie definicji matematycznej, np. stała 2 oznacza liczbę całkowitą (bez części ułamkowej) a stała 2.0 liczbę rzeczywistą (z zerową częścią ułamkową).

Zakres liczb rzeczywistych zależy od konkretnej architektury komputera i stosowanej reprezentacji liczb rzeczywistych.

Działania na liczbach całkowitych są dokładne (2+2 jest równe dokładnie 4), natomiast w przypadku liczb rzeczywistych wynik działania jest przybliżony (2.0+2.0 może być równe 4.000001).

Wspomnieliśmy o typie `complex`, podajmy zatem, jak wygląda zapis stałej tego typu. Składa się on z umieszczonych w nawiasach dwu liczb rzeczywistych oddzielonych przecinkiem. Pierwsza liczba to część rzeczywista; druga – część urojona liczby.

Np. `(0.0, 1.0)` to i
`(2.5, -1.0)` to $2.5 - i$
`(-1., 3.)` to $-1 + 3i$

2.3. Zmienne, specyfikacja typów zmiennych

Zmienna reprezentuje w programie Fortranowskim wielkość określonego typu; wartość zmiennej może być modyfikowana w trakcie działania programu. Zmienna jest identyfikowana przez swoją nazwę (zasady tworzenia nazw – patrz 1.1).

Poniższe nazwy zmiennych są poprawne:

```
x23, alpha, sloik, z, pi, j23, pipol, pole_kola
```

Poniższe nie (dlaczego?)

```
2x,  polekoła, dwa.i.pol
```

Zmienne mogą być dowolnego z typów omówionych wyżej.

Zasady określania jakiego typu jest dana zmienna w ogólnym przypadku są dosyć złożone. Zdecydowanie uprościmy sobie życie korzystając z dyrektywy

```
implicit none
```

Musi ona wystąpić na początku danej jednostki programu, przed innymi specyfikacjami.

Oznacza ona, że wszystkie zmienne w segmencie muszą być jawnie zadeklarowane z wyspecyfikowaniem typu (w przeciwnym wypadku kompilator zgłasza błąd). Częściowo zabezpiecza ona także przed błędami wynikającymi z omyłkowego przekręcenia nazwy zmiennej.

Specyfikacja typu zmiennej polega na podaniu nazwy typu oraz listy (oddzielonych przecinkami) nazw zmiennych, którym chcemy nadać ten typ, np.

```
integer liczba, n2, k
real r,pi,pole
complex calka
```

Przypomnijmy jeszcze raz, że wszystkie specyfikacje typów zmiennych muszą pojawić się przed pierwszą czynną instrukcją w segmencie.

2.4. Wyrażenia

Wyrażenie składa się ze stałych, zmiennych i wywołań funkcji połączonych operatorami i ewentualnie pogrupowanych za pomocą nawiasów.
Funkcje omówione zostaną bardziej szczegółowo w dalszej części tekstu, tutaj podajemy jedynie podstawowe informacje.

Do funkcji odwołujemy się przez jej nazwę, argument(y) funkcji podajemy w nawiasach () oddzielając przecinkami (jeśli jest więcej niż jeden).
Na tym etapie wymieńmy nazwy paru użytecznych dla nas funkcji wewnętrznych (będących częścią języka)(zestawienie funkcji wewnętrznych znajduje się w Uzupełnieniu A):

`abs` – wartość bezwzględna
`sqrt` – pierwiastek kwadratowy
`log` – logarytm naturalny
`log10` – logarytm dziesiętny
`exp` – funkcja wykładnicza
`sin, cos, tan` – funkcje trygonometryczne (argument w mierze łukowej)
`asin, acos, atan` – funkcje odwrotne do `sin, cos, tan`

Wszystkie wymienione wyżej funkcje mają jeden argument wywołania, np.

```
log(20.), sin(x), cos(Alpha), exp(-1.23), sqrt(4.)
```

Z wyjątkiem funkcji `abs` wszystkie wymienione wyżej nie dopuszczają argumentu całkowitego, zatem np. wywołanie `sqrt(4)` jest nieprawidłowe.

Operatory arytmetyczne to:

`+`	`-`	operator dodawania i odejmowania
`*`	`/`	operator mnożenia i dzielenia
`**`		operator potęgowania

Powyższe operatory są używane jako operatory dwuargumentowe, tzn. w wyrażeniach postaci

argument1 operator argument2

np. `2+5, 3*x, 2**3`.

Oprócz tego operatory + i – mogą wystąpić przed jednym argumentem, wtedy argument stojący przed operatorem przyjmuje się jako równy 0, np.

`-A` jest równoważne `0-A`
`+X` jest równoważne `0+X`

Podczas obliczania wartości wyrażenia arytmetycznego przestrzega się reguł wykonywania operacji w kolejności:
obliczanie wartości funkcji, potęgowanie, (mnożenie i dzielenie), (dodawanie i odejmowanie)

Kolejność tę można zmienić używając nawiasów okrągłych ( ). W przypadku konieczności użycia kilku poziomów nawiasów używa się tylko nawiasów okrągłych, np. `2*(a*(b+c)+1)`

Dla operatorów o jednakowym priorytecie obowiązuje kolejność „z lewa na prawo", z wyjątkiem potęgowania, które wykonuje się „od prawej do lewej" (czyli `x**y**2` to x^{y^2}).

Żadnego operatora nie można pominąć (zatem mamy `2*x` a nie `2x`), dwa operatory nie mogą występować obok siebie pod rząd (zatem `x**(-1)` a nie `x**-1`)

Przykłady wyrażeń arytmetycznych:

```
2
x
cos(x)      (pojedyncza stała, zmienna lub wywołanie funkcji to też wyrażenie
             arytmetyczne)
2+3
a+b
c/d + alpha/beta
x+y**2
2*cos(x2)+3*(z+2)
1+log(sqrt(23.))  (wywołania funkcji mogą się zagnieżdżać)
x**(1./3.)
```

Przy zapisie wyrażeń należy zwracać uwagę na kolejność wykonywania obliczeń.

Np. zapis `x**(1./3.)` to $x^{\frac{1}{3}}$, natomiast `x**1./3.` to po prostu x/3.

Podobnie zapis $\frac{a}{bc}$ jako `a/b*c` jest niepoprawny (oznacza bowiem $c\frac{a}{b}$)
Poprawny zapis to `a/(b*c)` lub `a/b/c`

2.5. Instrukcja przypisania

Jedną z najczęściej używanych w Fortranie instrukcji jest instrukcja przypisania.
Ma ona postać:

zmienna = *wyrażenie*

Podczas wykonania instrukcji przypisania najpierw oblicza się wartość wyrażenia po prawej stronie znaku = a następnie wartość ta zostaje przypisania zmiennej stojącej po lewej stronie znaku =.

Przykłady instrukcji przypisania:

```
n=10
x=3*y+6
z=cos(x+y)
```

Należy zwrócić uwagę na fakt, iż znak = w instrukcji przypisania nie jest operatorem relacji równości. Instrukcje

```
x=y
```

i

```
y=x
```

nie są równoważne. Po wykonaniu pierwszej z nich zmienne `x` i `y` będą miały taką wartość, jak początkowo miała zmienna `y`; po wykonaniu drugiej obie zmienne mają wartość równą początkowej wartości zmiennej `x`.

Zmienna może wystąpić po obu stronach instrukcji przypisania. W takim przypadku wyrażenie po prawej oblicza się dla starej wartości zmiennej, a następnie podstawia do zmiennej nową wartość, np.

```
n=3
n=n**2+n+1
```

Powyższa instrukcja nie jest zapisem równości, a zmianą wartości zmiennej `n`: po jej wykonaniu `n` będzie równe `13` ($3^2 + 3 + 1$).

Po lewej stronie instrukcji przypisania może wystąpić jedynie nazwa zmiennej; użycie stałej czy wyrażenia jest tam niedopuszczalne, np.:

```
3.=x+1          ! zle
x+y=5           ! tez zle
```

2.6. Konwersja typów

Omawiając wyrażenia i instrukcję przypisania nie zajmowaliśmy się dotąd kwestią typów stałych i zmiennych pojawiających się w wyrażeniach. Obecnie czas uzupełnić wiadomości na ten temat.

Dla ustalenia typu wyniku powstałego po działaniu operatora arytmetycznego stosujemy hierarchię typów (od najniższego do najwyższego):

```
integer, real, complex
```

Zasada ustalania typu wyniku jest następująca: jeśli oba argumenty operatora są jednakowego typu, to wynik będzie tego samego typu. Jeśli typ argumentów jest różny, to typ niższy zostaje skonwertowany na wyższy i wynik będzie tego typu (wyższego).
Konwersja ta polega na dodaniu zerowej części ułamkowej przy przejściu z `integer` do `real`, dodaniu zerowej części urojonej przy konwersji na liczbę zespoloną.

Przykład: Załóżmy, że zmienne `i,j` są typu `integer`, `x,y` typu `real` a `t` typu `complex`

w wyrażeniu `i+3` obie części składowe są typu `integer` i taki też będzie wynik

w wyrażeniu `x/j` wartość `j` zostanie skonwertowana z typu `integer` do `real`, wynik będzie typu `real`
w wyrażeniu `t+x` wynik będzie `complex` (`x` zostaje skonwertowane do `complex`)
podczas obliczania wyrażenia `t+x*i` najpierw `i` zostanie skonwertowane do `real` w celu obliczenia iloczynu `x*i`, następnie wynik tego mnożenia podniesiony do typu `complex` i taki będzie typ końcowego wyniku.

Z faktu, iż jednakowy typ argumentów operatora oznacza taki sam typ wyniku, wynikają istotne konsekwencje dotyczące działań na liczbach całkowitych.

Na przykład wynik dzielenia liczb całkowitych `1/3` jest całkowity, zostaje zatem zachowana tylko jego część całkowita równa 0. Podobnie `2**(-3)` jest też równe 0 (całkowite). W przypadkach tych następuje odrzucenie części ułamkowej (a nie zaokrąglenie), zatem wynikiem dzielenia `2/3` jest też 0 (całkowite).
W podanym przypadku zmianę typu wyniku możemy wymusić zmieniając typ któregokolwiek argumentu, zatem wynikiem działań `1./3` , `1.0/3.` , `1/3.0` będzie 0.333333
Sposobu tego nie można zastosować do wyrażenia typu `i/j` (`i./j` jest niepoprawne); w tym przypadku pomocne są funkcje konwersji typów (`int` na całkowity, `real` na rzeczywisty `cmplx` na zespolony); w naszym przykładzie możemy zapisać `real(i)/j` otrzymując w wyniku 0.3333333.

Pozostaje teraz ustalić, co stanie się jeśli typ wyrażenia po prawej stronie instrukcji podstawienia będzie inny niż typ zmiennej stojącej po lewej stronie. W takim przypadku dokonywana jest konwersja prawej strony na odpowiedni typ. Podniesienia wyniku do typu stojącego wyżej w hierarchii odbywa się w sposób opisany wyżej, konwersja na typ znajdujący się w hierarchii niżej polega na odrzuceniu zbędnej części urojonej liczby bądź części ułamkowej liczby (redukcja odbywa się przez obcięcie, a nie zaokrąglenie).

Przykłady:
Załóżmy, że zmienne `i` oraz `j` są typu `integer`, a zmienne `x` oraz `y` typu `real`.
W wyniku wykonania instrukcji

```
i=3.0
j=1.5
x=i/j
```

zmienna `x` przyjmie wartość 3.0 (rzeczywiste) ponieważ `j` jest równe 1 (całkowite)

Przy wykonaniu instrukcji

```
x=3.0
y=2.0
j=x/y
```

iloraz po prawej stronie instrukcji podstawienia zostanie obliczony jako 1.5 i skonwertowany do 1 w celu zapisania w zmiennej całkowitej.

2.7. Jeszcze trochę informacji o instrukcjach czytania i pisania...

W rozdziale 1.3 użyliśmy instrukcji `write` do wypisania pojedynczej stałej tekstowej. Instrukcji tej możemy podać listę wyrażeń oddzielonych przecinkami (w szczególności mogą to być stałe, zmienne, wywołania funkcji); podczas wykonywania instrukcji wartości wyrażeń są obliczane i wyprowadzane na ekran.

Na przykład instrukcja

```
write(*,*)'x=',x,' cos(x)=',cos(x)
```

wypisze na ekranie w jednym wierszu kolejno: tekst x= , aktualną wartość zmiennej `x` , tekst cos(x)= oraz wartość funkcji cos(x) obliczoną dla aktualnej wartości zmiennej `x`.

Nietrudno zauważyć, iż potrzebujemy instrukcji pozwalającej na podanie programowi wartości zmiennych będących danymi. Użycie w tym celu instrukcji podstawienia typu `x`=*wartość* jest niecelowe, ponieważ dowolna zmiana danych wymagałaby edycji kodu źródłowego programu i powtórnej kompilacji.

Nasz problem rozwiązuje instrukcja `read`. Używa ona listy zmiennych oddzielonych przecinkami, podczas wykonania oczekuje na podanie kolejnych wartości i przypisuje je kolejnym zmiennym na liście. Np.

```
read(*,*) x,y,n
```

podczas wykonania programu przypisze trzy podawane z klawiatury liczby kolejno zmiennym `x,y` oraz `n`. Parametr `(*,*)` oznacza, iż czytanie odbywa się z klawiatury w sposób „swobodny" tzn. nie wymagający szczególnego redagowania danych. Dane można podać w jednym wierszu oddzielając je przecinkami lub znakami tabulacji i kończąc wiersz naciśnięciem klawisza enter, lub podawać kolejno naciskając enter po wpisaniu każdej liczby.

Na liście instrukcji `read` mogą znaleźć się jedynie zmienne; nie mogą tam wystąpić stałe, wywołania funkcji czy wyrażenia (nie będące pojedynczymi zmiennymi).

Przy swobodnym czytaniu danych można podać liczbę całkowitą (np. 2) w momencie, gdy program oczekuje na wczytanie wartości zmiennej rzeczywistej (zostanie uzupełniona zerową częścią ułamkową). W przypadku odwrotnym (podanie liczby rzeczywistej, gdy program próbuje wczytać wartość zmiennej całkowitej) wygenerowany zostanie komunikat o błędzie.

Podczas wykonywania instrukcji `read` program oczekuje na podanie danych bez żadnych innych efektów. Korzystne jest zatem poprzedzić instrukcję `read` odnoszącą się do klawiatury instrukcją `write` wyświetlającą odpowiedni komunikat dla użytkownika, np.:

```
write(*,*) 'podaj wartosc x'
read(*,*) x
```

Użycie w wyrażeniu zmiennej, której wcześniej nie nadano wartości przez instrukcję `read` lub instrukcję przypisania spowoduje użycie wartości przypadkowo znalezionej w pamięci komputera, zatem wynik też będzie przypadkowy.

2.8. ... i możemy coś obliczyć.

Napiszmy program obliczający pole koła na podstawie zadanego promienia.

Potrzebujemy zmiennej rzeczywistej przechowującej wartość promienia (wczytamy tę wartość z klawiatury). Zmienne przechowujące wartości stałej π oraz pole koła też muszą być rzeczywiste.
Wybierzmy zatem nazwy zmiennych:

r – promień koła
pi – wartość pi
p – pole koła

Pozostaje kwestia określenia wartości zmiennej pi. Miłośnicy utworów typu „*kto z woli i myśli spamiętać pi zechce cyfry, ten zdoła*" ustalą, iż jest to ok. 3.1415926535, tym niemniej nie gwarantuje to użycia tylu cyfr znaczących, ile przechować może zmienna danego typu. Obliczmy więc pi np. z równości:

$\pi = 4\text{arctg}(1)$

(pamiętać należy o nazwie funkcji arctg w Fortranie i typie argumentu).

Podejmijmy jeszcze decyzję o jawnym specyfikowaniu zmiennych i ostatecznie mamy nasz przykładowy program:

```
program polekola
implicit none
real r,pi,p

write(*,*) 'podaj promien kola'
read(*,*) r
pi=4*atan(1.0e0)
p=pi*r**2
write(*,*) 'pole kola wynosi=',p
end program polekola
```

Ponieważ instrukcji `write` możemy podać wyrażenie, nasz program mógłby też wyglądać następująco:

```
program polekola
implicit none
real r

write(*,*) 'podaj promien kola'
read(*,*) r
write(*,*) 'pole kola wynosi=',4*atan(1.e0)*r**2
end program polekola
```

Zadanie:
Napisz program obliczający pole i objętość kuli na podstawie jej promienia.

3. Podejmowanie decyzji w programie

3.1. Relacje i wyrażenia logiczne

Kolejnym interesującym nas typem w Fortranie jest typ logiczny (`logical`). Wyrażenia logiczne mogą przyjmować dwie wartości: `.true.` i `.false.` (kropki są częścią zapisu stałej) – prawda i fałsz.

Najczęściej używanymi elementami wyrażeń logicznych są relacje. Przyjmują one wartości prawda lub fałsz w zależności od wartości porównywanych wyrażeń. Operator relacji działa na dwa argumenty numeryczne lub na dwa argumenty znakowe (tym przypadkiem nie będziemy się teraz zajmować):

argument1 operator_relacji argument2

Wynikiem takiej operacji jest jedna z dwu wartości logicznych.

Operatory relacji używane w Fortranie to:

`>`	- większe
`>=`	- większe lub równe
`<`	- mniejsze
`<=`	- mniejsze lub równe
`==`	- równe
`/=`	- różne

Argumenty arytmetyczne połączone operatorem relacji mogą być różnych typów; w takim przypadku następuje konwersja typów zgodnie z regułami opisanymi w r. 2.6.

Przykładowe zapisy relacji:

`2>3`	(czyli 2 > 3)
`2/=3`	(czyli 2 ≠ 3)
`x>=0`	(czyli x ≥ 0)
`n+m==5*n-1`	(czyli n+m = 5n+1)

Oczywiście wartość logiczna pierwszej z tych relacji to `.false.` a drugiej `.true.`; wartości logiczne pozostałych zależą od aktualnych wartości zmiennych `x`, `n` i `m`.

Wyrażenie logiczne w Fortranie może składać się ze stałych logicznych, zmiennych i funkcji typu logicznego oraz relacji połączonych operatorami logicznymi.

W Fortranie mamy następujące operatory logiczne:

`.not.`	- zaprzeczenie
`.and.`	- koniunkcja (iloczyn logiczny)
`.or.`	- alternatywa (suma logiczna)
`.eqv.`	- równoważność
`.neqv.`	- nierównoważność

Z wyjątkiem `.not.` operatory te są dwuargumentowe, tzn. używane są w wyrażeniach typu:

p operator q

Poniżej zestawione zostały wyniki tej operacji w zależności od wartości logicznej *p* i *q*.

p	*q*	.and.	.or.	.eqv.	.neqv.
.true.	.true.	.true.	.true.	.true.	.false.
.true.	.false.	.false.	.true.	.false.	.true.
.false.	.true.	.false.	.true.	.false.	.true.
.false.	.false.	.false.	.false.	.true.	.false.

Jednoargumentowy operator `.not.` zmienia wartość logiczną wyrażenia, przed którym stoi na przeciwną.

Priorytet działania operatorów logicznych (kolejność wykonywania operacji) maleje w kolejności: `.not.`, `.and.`, `.or.`, (równocenne `.eqv.` i `.neqv.`). W razie potrzeby kolejność wykonywania obliczeń można zmienić przy pomocy nawiasów ( ). W przypadku operatorów o jednakowym priorytecie obliczenia wykonuje się od lewej do prawej. Obliczanie wartości wyrażenia logicznego może być przerwane w momencie, kiedy znana jest jego wartość, np. obliczanie części składowych wyrażenia typu:

wyrazenie1 `.or.` *wyrazenie2* `.or.` *wyrazenie3* `.or.` *wyrazenie4*

może zostać przerwane, kiedy napotka się pierwsze wyrażenie prawdziwe (bo wtedy wiadomo już, że cała alternatywa jest prawdziwa).

Przy obliczaniu wartości wyrażenia logicznego operacje wykonuje się w kolejności:
obliczenie wyrażeń arytmetycznych, obliczenie wartości relacji, działanie operatorów logicznych.

Przykład:

Jeśli zmienna x ma wartość 1.0, zmienna y wartość 2.5, a `test` jest zmienną typu logicznego, to wartość wyrażenia

`x+2>y.and.x<0.or..not.y>=0` wynosi `.false.`

po podstawieniu wartości logicznych relacji jest ono bowiem równoważne wyrażeniu (nawiasami zaznaczono kolejność operacji przy uwzględnieniu priorytetu operatorów logicznych):

```
   (.true..and..false.).or.(.not..true.)
```

czyli `.false..or..false.`

zaś instrukcja

```
     test = x<y.neqv..false.
```

nadaje zmiennej `test` wartość `.true.`.

3.2. Porównywanie wartości typów rzeczywistych

Ze względu na skończoną reprezentację typów rzeczywistych w języku Fortran, wartości tych typów obarczone są pewnym błędem. Co więcej, pewnym (dodatkowym) błędem obarczony jest też wynik każdej operacji arytmetycznej wykonywanej na tych wartościach.

Wskutek tego dwie liczby, które byłyby sobie równe, gdyby obliczenia nie były obciążone błędami reprezentacji, mogą nie być równe w przypadku przeprowadzania obliczeń na wartościach typów rzeczywistych. Analogicznie, dwie liczby które różniłyby się od siebie nieznacznie w przypadku przeprowadzenia dokładnych obliczeń, mogą być równe sobie, gdy obliczenia przeprowadzamy na wartościach typów rzeczywistych.

Powoduje to, że używanie operatora równości (==) i nierówności (/=) do porównania dwóch wartości typu rzeczywistego, jakkolwiek dopuszczalne składniowo, stanowi zwykle błąd logiczny w programie. Ponieważ błąd ten wynika ze specyficznych właściwości relacji równości/nierówności, nie dotyczy on innych relacji pomiędzy wartościami typów rzeczywistych.

Problem ten nie występuje w przypadku porównywania wartości typów całkowitych.

Zauważmy, że sprawdzanie, czy dwie wartości rzeczywiste są sobie równe, to na ogół złe postawienie problemu. Wartości te bowiem zwykle reprezentują wielkości, które w praktyce możemy zmierzyć tylko z pewną skończoną dokładnością, np. długość, masę, temperaturę, gęstość elektronową, itp. Wobec tego zamiast pytać, czy dwie wielkości rzeczywiste są równe, powinniśmy zapytać, czy różnica pomiędzy nimi jest tak mała, że możemy ją zaniedbać. Zatem, jeżeli z rozważenia fizycznych uwarunkowań problemu wynika, że dwie wielkości różniące się np. o 1×10^{-15} można uznać za praktycznie równe, to zamiast relacji:

```
x==y
```

zapiszemy

```
abs(x-y)<1.e-15
```

(użyliśmy funkcji zwracającej wartość bezwzględną, gdyż nie wiemy z góry, jaki jest znak różnicy).

3.3. Instrukcja blokowa `if`

Wiedząc jak ustalić wartość logiczną wyrażenia, możemy zastosować tę wiedzę do podejmowania decyzji w programie.

Jeśli w zależności od prawdziwości pewnych warunków logicznych zmieniają się czynności, które powinien wykonać program, możemy posłużyć się blokową instrukcją `if`. Jej ogólna postać to:

```
if (warunek) then
   ciąg instrukcji 1
else
   ciąg instrukcji 2
end if
```

(dopuszczalna jest pisownia `end if` i `endif`)

W powyższym schemacie *warunek* oznacza wyrażenie logiczne (w szczególności może to być relacja). Jeśli jest ono prawdziwe, to wykonywany jest *ciąg instrukcji 1* (tzw. blok if). Jeśli wyrażenie jest fałszywe, to wykonywany jest *ciąg instrukcji 2* (tzw. blok else). W obu przypadkach następnie wykonywana jest instrukcja następująca po `end if`.

Przykład:

```
if (x+y>0) then
   z=2.5
else
   z=-2.5
end if
write(*,*) z
```

W wyniku wykonania powyższej instrukcji na ekranie komputera pojawi się liczba 2.5 jeśli suma `x+y` była większa od 0 albo liczba –2.5 jeśli suma `x+y` była mniejsza lub równa 0. Zwróć uwagę na wcięcie instrukcji bloków if i else (przesunięcie w prawo). Nie jest ono wymagane, jednak zaleca się stosowanie wcięć w instrukcjach blokowych, gdyż poprawia to czytelność kodu.

Część `else` można pominąć – konstrukcja wygląda wtedy następująco:

```
if (warunek) then
   ciąg instrukcji
end if
```

Ciąg instrukcji jest wykonywany, gdy warunek jest spełniony, w przeciwnym wypadku przechodzi się od razu do instrukcji następującej po `end if`.

Instrukcje blokowe `if` mogą być zagnieżdżone – jak w przykładzie:

```
if (x>0) then
    write(*,*) 'x wieksze od 0'
else
    if (x<0) then
        write(*,*) 'x mniejsze od 0'
    else
        write(*,*) 'x rowne 0'
    end if
end if
```

W przykładzie tym część `else` zewnętrznej instrukcji `if` jest wykonywana, gdy `x` jest mniejsze lub równe 0, stąd użycie drugiej instrukcji `if` do sprawdzenia, z którą z tych możliwości mamy do czynienia.

Zadanie: wykorzystując powyższy szkielet dwu instrukcji blokowych `if`, napisz program, który policzy pierwiastki rzeczywiste równania kwadratowego o zadanych współczynnikach a,b,c lub stwierdzi, że pierwiastki rzeczywiste nie istnieją.

4. Organizacja obliczeń cyklicznych

4.1. Cykl typu „podczas gdy”

Implementacja algorytmów wymaga często zaprogramowania powtarzających się czynności. Z punktu widzenia organizacji programu, tego typu powtórzenia możemy podzielić na dwie grupy:
1) Ilość powtórzeń nie jest z góry znana – powtarzać cykl instrukcji musimy tak długo, jak długo są spełnione pewne warunki. Ten typ omówimy w niniejszym podrozdziale.
2) Ilość powtórzeń jest z góry zadana – tego typu powtórzenia omówimy w podrozdziale 4.2.

Cykl „podczas gdy” realizujemy przy pomocy konstrukcji „`do while`”. Jej ogólna postać jest następująca:

```
do while (warunek)

  ciąg instrukcji

end do
```

Jeżeli *warunek* jest spełniony (tzn. jego wartość logiczna to `.true.`) wykonywany jest ciąg instrukcji między `do` a `end do`, po czym następuje kolejne sprawdzenie warunku i ewentualnie kolejne wykonanie ciągu instrukcji. Jeśli warunek nie jest spełniony (`.false.`), konstrukcja kończy pracę i wykonywana jest instrukcja następna po `end do`. Innymi słowy, ciąg instrukcji powtarzany jest tak długo, jak długo spełniony jest warunek. Możliwa jest sytuacja, kiedy ciąg instrukcji nie będzie wykonany ani razu (w przypadku, kiedy warunek od początku nie był spełniony).

Zwróćmy uwagę, że instrukcje zawarte w pętli muszą powodować po pewnej liczbie powtórzeń zmianę wartości logicznej warunku, w przeciwnym wypadku program wpadnie w niekończącą się pętlę.

Przykład: Dodatni pierwiastek równania $x^2 = c$ można wyznaczyć iteracyjnie, przyjmując jakąś startową wartość x_0 a następnie obliczając kolejne przybliżenia:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{c}{x_n} \right)$$

Ciąg obliczanych wartości dąży do $\sqrt{c}$.

Spróbujmy zaprogramować ten algorytm, startując z 1 jako pierwszego przybliżenia i prowadząc obliczenia tak długo, jak długo dwa kolejne przybliżenia różnią się o więcej niż pewną zadaną wartość ε. Nie potrafimy z góry powiedzieć, ile iteracji będziemy musieli wykonać.

Potrzebne dane to zatem *c* i ε.

Zauważmy, że nie musimy w programie pamiętać wszystkich obliczanych kolejno wartości – do sprawdzenia warunku wystarczy nam znajomość przedostatniej (oznaczmy ją `x`) i ostatniej (oznaczmy ją `xn`).

Nasz program może więc wyglądać następująco:

```
program pierw
implicit none

real c,eps,x,xn

write(*,*)'podaj pierwiastkowana liczbe i dokladnosc'
read(*,*) c,eps

x = 1.0
xn=(x+c/x)/2          ! obliczamy pierwsze przybliżenie
write(*,*) xn

do while (abs(xn-x)>eps)  ! rozpoczynamy iteracje
   x=xn
   xn=(x+c/x)/2
   write(*,*) xn
end do

write(*,*) 'c= ',c
write(*,*) 'wartosc pierwiastka z c'
write(*,*) 'z iteracji = ',xn,' z funkcji= ',sqrt(c)

end program pierw
```

Wśród szeregu powtarzanych instrukcji umieściliśmy instrukcję `write` wypisującą aktualnie obliczone przybliżenie, aby móc obserwować zbieżność. W warunku instrukcji `do while` badamy wartość bezwzględną różnicy dwu ostatnich przybliżeń, ponieważ nie wiemy z góry, jaki jest jej znak. Jeśli warunek jest spełniony (czyli mamy kontynuować iteracje), to ostatnio obliczoną wartość (`xn`) przypisujemy zmiennej `x` obliczamy następne przybliżenie. Zatem po każdym wykonaniu cyklu instrukcji w pętli, zmienna `xn` zawiera ostatnie a zmienna `x` – przedostatnie obliczone przybliżenie pierwiastka z *c*.
Przy podawaniu końcowego wyniku wypisaliśmy dla porównania wartość dokładną obliczoną przy pomocy standardowej funkcji Fortranu. Należy zwrócić uwagę na to, iż jeśli dwie kolejno obliczone przybliżenia różnią się o mniej niż ε, to wcale nie musi stąd wynikać, iż różnica w stosunku do wartości dokładnej też jest mniejsza niż ε.

Zadanie. Zmodyfikuj program tak, aby móc zmieniać startowe przybliżenie pierwiastka (w naszym programie było równe 1, niezależnie od *c*). Zbadaj, jak wpływa ono na zbieżność.
Zbadaj, czy program daje poprawne wyniki dla dużych (np. 10^{20}) i małych (np. 10^{-20}) *c*. Jeśli nie, zastanów się, czy potrafisz zaproponować jakąś poprawkę.

4.2. Instrukcja `do`

Drugim przypadkiem obliczeń cyklicznych jest sytuacja, w której znamy z góry ilość powtórzeń. Przykładem może być wypisanie tabeli wartości funkcji w zadanym przedziale, czy obliczenie sumy liczb od 1 do *N*.

Możemy wtedy skorzystać z instrukcji `do` w postaci:

```
do zmienna = w_pocz, w_konc, skok
  (instrukcje)
end do
```

zmienna oznacza tutaj nazwę tzw. zmiennej sterującej.
w_pocz, *w_konc* i *skok* są wyrażeniami arytmetycznymi (w szczególności mogą to być stałe lub nazwy zmiennych)
Jeśli *skok* jest równy 1, to można go pominąć.

Wykonanie pętli przebiega następująco (jest to opis praktycznych skutków działania, a nie technicznej realizacji pętli):

Początkowo zmienna sterująca *zmienna* przyjmuje wartość początkową *w_pocz*. Sprawdzany jest następnie warunek, czy wykonywać instrukcje pętli, tzn. czy *zmienna* jest nie większa od wartości końcowej *w_konc*, jeśli *skok* jest dodatni, albo czy *zmienna* jest nie mniejsza od *w_konc*, jeśli *skok* jest ujemny. Jeśli warunek jest spełniony, to wykonywany jest ciąg instrukcji między `do` a `end do` (dopuszczalna pisownia z odstępem lub bez). Następnie wartość zmiennej sterującej *zmienna* zmienia się o *skok* i warunek sprawdza się ponownie. W momencie, kiedy warunek nie jest spełniony, działanie pętli kończy się i wykonywana jest instrukcja następna po `end do`.

Instrukcje wewnątrz pętli nie mogą powodować zmiany wartości zmiennej sterującej.

Przykłady:
a)

```
do i = 1,5,1
     (...)
end do
```

Instrukcje w pętli będą wykonane 5 razy (zmienna sterująca `i` przyjmie kolejno wartości 1,2,3,4 i 5). Ponieważ skok jest równy 1, więc można go pominąć:

```
do i = 1,5
     (...)
end do
```

b)

```
do kk = 10,20,4
     (...)
end do
```

Instrukcje w pętli będą wykonane 3 razy (zmienna sterująca `k` przyjmie kolejno wartości 10, 14, 18).

c)

```
do licznik = 20,-3,-5
     (...)
end do
```

Instrukcje w pętli będą wykonane 5 razy (zmienna sterująca `licznik` przyjmie kolejno wartości 20, 15, 10, 5, 0).

d)

```
do j = 10,10,10
     (...)
end do
```

Instrukcje w pętli będą wykonane tylko 1 raz (zmienna sterująca `j` przyjmie wartość 10).

e)

```
do j = 20,10,2
     (...)
end do
```

Instrukcje w pętli nie będą wykonane ani razu (bo warunek wykonywania pętli jest niespełniony już na starcie: wartość początkowa jest większa od końcowej przy dodatnim skoku).

Łatwo zauważyć, że *skok* nie może być równy 0.

Pętle mogą się zagnieżdżać (wtedy muszą mieć różne zmienne sterujące):

```
n = 4
do i = 1,n

     do j = 1,5,2

     end do

end do
```

Szybciej wykonywana jest pętla wewnętrzna. Zatem w powyższym przykładzie w trakcie przebiegu pętli zewnętrznej dla `i` = 1 zostanie 3 razy wykonana pętla wewnętrzna (dla `j` równego kolejno 1, 3 i 5). Podobnie będzie dla kolejnych wykonań pętli zewnętrznej (dla `i` równego 2, 3 i 4).

Ze względu na specyfikę obliczeń na liczbach rzeczywistych powinno się używać całkowitych zmiennych sterujących pętlą oraz wyrażeń *w_pocz*, *w_konc* i *skok*. W nowszych standardach Fortranu wartości rzeczywiste nie będą dopuszczalne.

Wspomnijmy tu o dwu instrukcjach pozwalających na przerwanie wykonywania pętli:

Instrukcja `exit` powoduje przerwanie wykonywania pętli i przejście do wykonywania instrukcji następnej po `end do`.
Instrukcja `cycle` powoduje przerwanie bieżącego cyklu wykonywania pętli i przejście do następnego, tzn. po wykonaniu instrukcji `cycle`, zostają pominięte instrukcje, które po niej następują, aż do `end do`, zmienna sterująca przyjmuje kolejną wartość i zaczyna się od początku następny obrót pętli (chyba, że przerwany cykl był ostatnim).

4.3. „Nieskończona" pętla `do`

Możliwe jest zastosowanie następującej odmiany pętli `do`:

```
do
   ciąg instrukcji
end do
```

Jak widać, nie występuje tu zmienna sterująca, ani parametry określające ilość powtórzeń. Aby powtarzanie takiej pętli zostało zatrzymane, w ciągu instrukcji musi pojawić się instrukcja, która przerwie wykonywanie pętli, np. instrukcja `exit`.

Zobaczmy na przykładzie, jak moglibyśmy wykorzystać tę konstrukcję zamiast `do while` w programie do obliczania pierwiastka.

```
program pierw
implicit none

real c,eps,x,xn

write(*,*)'podaj pierwiastkowana liczbe i dokladnosc'
read(*,*) c,eps

x = 1.0

do
   xn=(x+c/x)/2
   write(*,*) xn
   if (abs(xn-x)<=eps)  then  ! osiagnelismy zadana dokl.
     exit
   end if
   x=xn
end do

write(*,*) 'c= ',c
write(*,*) 'wartosc pierwiastka z c'
write(*,*) 'z iteracji = ',xn,' z funkcji= ',sqrt(c)

end program pierw
```

Zwróć uwagę, że odwróciliśmy sprawdzany warunek (dlaczego?).

4.4. Przykłady wykorzystania pętli do

1) Zróbmy tabelę wartości funkcji z = x^2y dla x i y zmieniających się w przedziałach od –1 do 1 z krokiem 0.1

Aby w pętli móc używać całkowitej wartości kroku, przeskalujmy jej granice i skok, tzn. organizujmy pętlę od –10 do 10 z krokiem 1. Musimy wtedy uwzględnić przeskalowanie przy obliczaniu funkcji, (podzielić wartość argumentu przez 10):

```
program tabelka
implicit none

integer ix,iy
real x,y

do ix = -10,10
     x = ix/10.
     do iy = -10,10
       y = iy/10.
       write(*,*) 'x=',x,' y=',y,' z=', x**2*y
     end do
end do
end program tabelka
```

Zadanie. W powyższym przykładzie wiedzieliśmy z góry, jak przeskalować zmienne, aby móc używać całkowitych wartości do indeksowania pętli. Zmodyfikuj program tak, aby użytkownik mógł podać granice zmienności x i y oraz krok (kroki).
Wskazówka: Jeśli zmienna rzeczywista x ma zmieniać się od xp do xk z krokiem xs, to możemy zapisać następującą konstrukcję pętli:

```
do i = 0,nx
     x = xp + i*xs

end do
```

Zastanów się, jak ze znajomości `xp`, `xk` i `xs` wyznaczyć wartość `nx` potrzebną do zorganizowania tej pętli. Pomocna może być funkcja `nint()` zwracająca najbliższą liczbę całkowitą (np. `nint(4.7)` jest równe 5).

2) Obliczmy sumę kolejnych liczb całkowitych od 1 do *N*.

Program może wyglądać następująco:

```
program sumaliczb
implicit none

integer   i,s,n

write(*,*) 'podaj N'
```

```
read(*,*) n

s = 0
do i = 1,n
    s = s + i
end do

write(*,*) 'suma liczb od 1 do ',n,' wynosi ',s
end program sumaliczb
```

W powyższym programie zmienna `s` przechowuje wartość sumy, dlatego przed rozpoczęciem sumowania musi zostać wyzerowana. Pętla do przebiega po kolejnych liczbach naturalnych od 1 do N. Przy każdym przebiegu pętli suma jest zwiększana o kolejną liczbą (tzn. `i`).

Zadania. Napisz program, który policzy sumę kwadratów kolejnych liczb naturalnych od 1 do N.

Napisz program, który policzy silnię zadanej liczby. Co należy zrobić ze zmienną przechowującą wartość iloczynu przed rozpoczęciem wykonywania pętli? Sprawdź, jakie wyniki daje program dla n = 12, 13, 14, 15, 16, 17. Takie zachowanie przy przekroczeniu zakresu dopuszczalnych liczb całkowitych jest normalne – wynika z własności arytmetyki na tych liczbach. Zmień wobec tego typ zmiennej przechowującej wartość silni na `real` i sprawdź, dla jak dużego n możesz obliczyć $n!$

Dla dociekliwych:
Wspomnieliśmy w rozdziale 2, iż standard określa, że powinny być dostępne co najmniej dwa rodzaje liczb rzeczywistych. Możemy wobec tego spodziewać się, iż mamy dostępny jeszcze jakiś rodzaj, dający nam większy zakres liczb. Musimy tylko wiedzieć, jak zadeklarować zmienną tego typu. Wykorzystamy w tym celu funkcję Fortranu o nazwie `selected_real_kind` . Nie wnikając w szczegóły zobaczmy, jak może wyglądać deklaracja zmiennej o nazwie `s`:

```
real(selected_real_kind(10,2000)) s
```

Wartości w nawiasach określają, iż zażądaliśmy od kompilatora, aby zmienna `s` była rzeczywista i mogła przechować wartości rzędu co najmniej 10^{2000} z dokładnością 10 cyfr znaczących. Jeśli żądany przez nas rodzaj zmiennej nie jest osiągalny, kompilator wypisze odpowiedni komunikat o błędzie. W przypadku kompilatora gfortran mamy taki rodzaj zmiennej rzeczywistej do dyspozycji; zbadaj, dla jak dużych n można policzyć silnię z jego użyciem.

Dla jeszcze większych n pozostaje obliczanie $\log n!$. Napisz odpowiedni program.

5. Tablice

5.1. Deklaracje tablic

Tablica jest uporządkowanym zbiorem obiektów tego samego typu opatrzonych wspólną nazwą. Poszczególne obiekty składowe wskazujemy za pomocą indeksów. W zależności od liczby indeksów będziemy mieć tablice jedno- (odpowiednik wektorów), dwu- (odpowiednik macierzy) lub więcej wymiarowe. Typ elementu tablicy może być jednym z typów języka Fortran, możemy więc mieć tablice o elementach całkowitych, rzeczywistych, logicznych, itp. Zakres zmienności indeksów określa kształt tablicy oraz jej całkowity rozmiar. Np. tablica o dwu wierszach i trzech kolumnach jest dwuwymiarowa (potrzebujemy dwu indeksów do określenia jej elementów), jej kształt to 2×3 a rozmiar jest równy 6.

Przed użyciem tablice muszą być zadeklarowane. Przy deklaracji możemy określić rozmiar tablicy (tablice statyczne) lub podać tylko liczbę wymiarów, a rozmiar określić później (tablice alokowalne).

Deklarator tablicy statycznej ma postać:

nazwa(*zakres1*,*zakres2*,...)

gdzie *nazwa* to nazwa tablicy a *zakres1*, *zakres2*, ... określają zakresy zmienności poszczególnych indeksów. Zakresów tych podaje się tyle, ile indeksów chcemy używać do numerowania elementów tablicy (maksymalnie może być ich 7).

Zakres zapisujemy w sposób następujący:
gd:*gg* , gdzie *gd* to dolny zakres indeksu, a *gg* – górny
lub
gg , *gdzie gg* to górny zakres indeksu (dolny jest wtedy równy 1)
gd i *gg* muszą być wyrażone przez stałe całkowite

Deklarator tablicy alokowalnej wygląda następująco:

nazwa(:,:,...)

Podaje się tyle dwukropków, ile ma być indeksów. Ich zakres zmienności (rozmiar tablicy) będzie można określić później.

Deklarator tablicy (lub listę deklaratorów oddzielonych przecinkami) można użyć po specyfikacji typu zmiennej (zadeklarowany zostaje wtedy rozmiar tablicy i typ jej elementów).

Na razie będą nas interesować dwa sposoby deklarowania tablicy:
- z jawną specyfikacją rozmiaru (wtedy podajemy zakresy zmienności indeksów), np.

```
real x(0:100), y(2,5)
real ty(3,3,3)
integer liczniki(20)
```

- jako tzw. tablicę alokowalną – przy deklaracji określamy tylko liczbę wymiarów (przy pomocy dwukropków); jej kształt i rozmiar będziemy mogli określić później, np:

```
real, allocatable :: tx(:,:), z(:)
```

Zwróćmy uwagę na szczegóły powyższej deklaracji. Po określeniu typu elementów tablicy, wyspecyfikowaliśmy, iż tablica ma mieć atrybut `allocatable`. Jeśli określamy atrybuty zmiennej(ych) (`allocatable` to tylko jeden z możliwych), to listę nazw musimy poprzedzić parą dwukropków (`::`). Ta para znaków może zawsze wystąpić w deklaracji zmiennej, ale ponieważ nie jest obowiązkowa gdy nie specyfikujemy atrybutów, to dotąd ją pomijaliśmy.
Nie podaliśmy zakresu zmienności indeksów, ale określiliśmy wymiar tablic (określa go liczba dwukropków w nawiasach)

W powyższych przykładach zadeklarowano:
- jednowymiarową tablicę `x` o elementach rzeczywistych indeksowanych od 0 do 100
- tablicę `y` o elementach rzeczywistych indeksowanych dwoma indeksami zmieniającymi się od 1 do 2 i od 1 do 5
- trójwymiarową tablicę `ty` o elementach typu `real`; wszystkie trzy wskaźniki zmieniają się od 1 do 3
- jednowymiarową tablicę `liczniki` o elementach całkowitych numerowanych od 1 do 20
- dwuwymiarową tablicę `tx` o elementach rzeczywistych oraz jednowymiarową tablicę `z` także o elementach rzeczywistych, zakresy zmienności indeksów (a co za tym idzie kształt i rozmiar tablicy będziemy mogli określić później).

Żeby móc do czegoś wykorzystać tablicę alokowalną musimy ją alokować (tzn. w pamięci komputera musi zostać przydzielony na nią odpowiedni obszar). Wykorzystujemy do tego polecenie `allocate` w postaci:

`allocate`(*lista_deklaratorów_tablic*)

np. tablice zadeklarowane wyżej możemy zalokować poleceniami:

```
allocate(tx(0:10,10:20))
allocate(z(n))
```

albo obie równocześnie:

```
allocate(tx(0:10,10:20),z(n))
```

Liczba wymiarów tablicy użyta przy alokacji musi zgadzać się z użytą podczas deklaracji.
Zwróćmy uwagę, że przy alokacji tablicy `z` użyto zmiennej `n` do określenia zakresu indeksów. Ogólnie można tu użyć wyrażenia całkowitego. Podanie zakresu poprzez zmienną lub wyrażenie to typowy przykład użycia tablic alokowalnych. Stosujemy go w przypadku, kiedy dopiero w trakcie wykonywania programu znana jest informacja o potrzebnym rozmiarze tablicy (wynikającym np. z ilości wczytanych danych).

Deklaracje tablic z jawnym określeniem rozmiaru stosujemy najczęściej w przypadku, kiedy potrzebne rozmiary są znane już na etapie pisania programu i nie będą zależały od okoliczności wykonania programu; np. przekształcenia współrzędnych w przestrzeni

trójwymiarowej opisane są macierzami 3×3, zatem w programie dokonującym takich transformacji z góry wiadomo, że potrzebne będą tablice 3×3.

W chwili, kiedy dane zapisane w alokowalnej tablicy nie będą już więcej potrzebne, tablicę możemy dealokować, zwalniając pamięć komputera. Służy do tego polecenie `deallocate`:

```
deallocate(lista_nazw_tablic)
```

np.:

```
deallocate(tx,z)
```

Nie można zmienić rozmiarów już alokowanej tablicy bez jej dealokacji i powtórnego alokowania a podaniem nowego rozmiaru.

5.2. Elementy i wycinki tablic. Użycie tablic w wyrażeniach i instrukcji przypisania.

Element tablicy zwany jest często zmienną indeksowaną. Odwołanie do konkretnego elementu tablicy polega na podaniu nazwy tablicy i wartości indeksów określających ten element. Wartości te podaje się jako wyrażenia całkowite; musi ich być tyle, ile wymiarów tablicy zadeklarowano.

Przykłady użycia elementów tablic zadeklarowanych w p. 5.1:

```
read(*,*)x(0),x(20)
y(1,3) = y(1,1) * ty(1,1,3)
alpha = acos(x(10))
write(*,*) liczniki(1),liczniki(2)
```

Kolejność rozmieszczenia elementów tablicy w pamięci komputera odpowiada najszybszej zmianie indeksu pierwszego, potem drugiego, itd.

Na przykład elementy tablicy

```
real tab(3,3)
```

umieszczone są w pamięci w kolejności

```
tab(1,1),tab(2,1),tab(3,1),tab(1,2),tab(2,2),tab(3,2),tab(1,3,
tab(2,3),tab(3,3)
```

(Mówimy często, że tablice dwuwymiarowe zapamiętywane są kolumnami).

W Fortranie 90 można używać także wycinków tablic. Wycinany zakres tablicy możemy określić podając dla danego wymiaru zamiast indeksu tzw. tryplet w postaci: *dz*:*gz*:*sk*, gdzie *dz* i *gz* to odpowiednio dolny i górny zakres zmienności wskaźnika a *sk* to krok, z jakim jest zmieniany. Jeśli krok jest równy 1, można go pominąć; pominięcie *dz* lub *gz* oznacza użycie wartości z jaką zadeklarowano lub alokowano tablicę.
Przykład: Jeśli tablica `tab` zadeklarowana została jak wyżej, to:

`tab(1,1:2)` jest 1-wymiarowym wycinkiem składającym się z elementów (1,1) i (1,2) tablicy źródłowej

`tab(1:2,:)` jest 2-wymiarowym wycinkiem składającym się z elementów (1,1), (2,1), (1,2), (2,2), (1,3), (2,3)

W konstrukcjach typu:

c = a *operator* b

c = *funkcja*(a)

a,b,c mogą być tablicami (nie wszystkie funkcje mogą być użyte w takiej instrukcji, patrz Dodatek A). Muszą być wtedy o zgodnych rozmiarach (tzn. tego samego kształtu). Działanie jest wtedy wykonywane na odpowiadających sobie parach elementów tablic a jego wynik zapisany jest w odpowiednim elemencie tablicy po lewej stronie znaku =. Podobnie w drugim przypadku, obliczenie wartości funkcji następuje dla poszczególnych elementów tablicy.
Jednym z argumentów działania może być zmienna skalarna (skalar jest zgodny rozmiarowo z dowolną tablicą) i wtedy jej wartość będzie użyta w działaniu na każdym elemencie tablicy.

Przykład. Przy deklaracjach:

```
real a(10),b(10),c(10)
real d
```

poprawne są instrukcje:

```
a=1.0
```
(przypisanie wartośći 1.0 każdemu elementowi tablicy `a`)

```
c=a+b
```
(elementy tablicy `c` są sumą odpowiadających sobie elementów tablic `a` i `b`)

```
c=a*b
```
(elementy tablicy `c` są iloczynami odpowiadających sobie elementów tablic `a` i `b`)

```
c=1./a
```
(elementy tablicy `c` są odwrotnościami elementów tablicy `a`)

```
b=d*a
```
(elementy tablicy `b` są iloczynami elementów tablicy `a` i liczby `d`)

```
c=cos(a)
```
(elementy tablicy `c` są wartościami cosinusów odpowiednich elementów tablicy `a`)

Zwróćmy uwagę, że instrukcje `c=a+b` i `b=d*a` odpowiadają sumie macierzy i iloczynowi macierzy przez skalar, natomiast polecenia `c=a*b` czy `c=1./a` nie są zapisem mnożenia macierzy lub obliczania jej odwrotności.

Ponieważ wycinek tablicy też jest tablicą, może być używany w instrukcjach opisanych wyżej.

5.3. Przykład użycia tablic

Wektor w przestrzeni *N*-wymiarowej określamy przez podanie jego *N* składowych:

$$\vec{x} = (x_1, x_2, \ldots, x_N)$$

Długość wektora możemy obliczyć ze wzoru:

$$|\vec{x}| = \sqrt{\sum_{i=1}^{N} x_i^2}$$

Napiszmy program, który dla zadanego *N* wczyta składowe wektora (zapamiętując je jako elementy jednowymiarowej tablicy) i policzy jego długość.

Użyjemy tablicy alokowalnej – gdy program dowie się, ile wynosi *N* w konkretnym przypadku alokuje odpowiednio dużą tablicę.

```
program wektorek
implicit none

real, allocatable :: x(:)
real dlugosc
integer i,n

write(*,*) 'podaj liczbe skladowych wektora'
read(*,*) n

allocate(x(n))

write(*,*) 'podawaj kolejno skladowe wektora'

do i=1,n
     read(*,*) x(i)
end do

dlugosc=0

do i=1,n
   dlugosc = dlugosc + x(i)**2
end do

deallocate(x)

dlugosc=sqrt(dlugosc)

write(*,*) 'dlugosc wektora = ',dlugosc
end program wektorek
```

Pierwsza pętla wczytuje kolejno *N* liczb zapamiętując je jako kolejne elementy tablicy x. Druga pętla sumuje kwadraty składowych, zatem po zakończeniu jej działania zmienna `dlugosc` przechowuje kwadrat długości – dopiero w następnej instrukcji zostaje obliczony pierwiastek z sumy kwadratów.

Zadanie. 1) Rozbuduj program tak, aby przeczytał składowe dwu wektorów, policzył ich iloczyn skalarny oraz ich długości a następnie kąt między wektorami.

2) Powyższy przykład (także po rozbudowie z pkt. 1) można napisać tak, aby w ogóle nie używał tablic. Zastanów się, jak to zrobić.

5.4. Implikowana instrukcja `do`

Załóżmy, że w naszej walce z tablicami doszliśmy do etapu użycia tablic dwuwymiarowych (odpowiednik macierzy). Zobaczmy, jak moglibyśmy wczytać elementy takiej tablicy:

```
implicit none
real, allocatable :: x(:,:)
integer nw,nk,i,j

write(*,*) 'podaj liczbe wierszy i kolumn macierzy'
read(*,*) nw,nk
allocate(x(nw,nk))
write(*,*)'podawaj pojedynczo elementy macierzy'

do i=1,nw
    do j=1,nk
         read(*,*) x(i,j)
    end do
end do
```

W powyższym przykładzie użytkownik programu musi podawać po kolei elementy tablicy (wierszami) – naciskając enter po wpisaniu każdej liczby. Nie jest to sposób wygodny. Znacznie lepiej byłoby, gdyby można było podawać całe wiersze macierzy. Pomóc może w tym tzw. implikowana instrukcja `do` (dopuszczalna w ramach instrukcji `read` lub `write`). Instrukcja ta ma postać:

(*lista, zmienna = w_pocz, w_konc, skok*)

Lista to lista obiektów do wypisania/wczytania przez instrukcję `write/read`; *zmienna, w_pocz, w_konc, skok* mają takie samo znaczenie, jak w instrukcji `do`.
Instrukcja `read` lub `write` powtórzy czytanie lub wypisywanie obiektów z listy tyle razy, ile wynika z wartości wyrażeń określających pętlę.

W naszym przypadku moglibyśmy zapisać

```
do i=1,nw
   read(*,*) (x(i,j), j=1,nk)
end do
```

Wewnętrzna pętla `do` została zastąpiona przez `do` implikowane w instrukcji `read`. Ponieważ skok jest równy 1, został pominięty w zapisie pętli.
Dane można teraz podawać wierszami (po `nk` danych w wierszu).

Analogicznie może wyglądać implikowane `do` w instrukcji `write`.

```
do i=1,nw
   write(*,*) (x(i,j), j=1,nk)
end do
```

Każda instrukcja `write` wypisze `nk` elementów, tzn. cały wiersz macierzy.
Implikowanym do moglibyśmy zastąpić także pętlę po zmiennej `i`:

```
write(*,*) ((x(i,j), j=1,nk),i=1,nw)
```

(zwróć uwagę na zagnieżdżenia nawiasów).
Taka konstrukcja oznaczałaby jednak wypisanie wszystkich elementów tablicy dwuwymiarowej w jednym wierszu – co nie jest chyba zbyt rozsądne.
W instrukcji `write` lub `read` można użyć nazwy tablicy, bez podania konkretnego jej elementu, np.:

```
real, allocatable :: x(:,:)

allocate(x(nw,nk))
    ....
read(*,*) x
    ....
write(*,*) x
```

W takim przypadku wypisywane (wczytywane) są wszystkie elementy tablicy (tyle, ile wynika z deklaracji rozmiarów, czyli w naszym przypadku `nw` × `nk`), w kolejności, w jakiej rozmieszczone są w pamięci komputera (patrz 5.2).

Zadanie. Napisz program, który przeczyta dwie macierze prostokątne (o zadanych rozmiarach) i policzy ich sumę. Wykorzystaj implikowane `do` do wprowadzania/wyprowadzania danych.

6. Operacje wejścia/wyjścia. Używanie plików zewnętrznych.

W rozdziale tym uzupełnimy wiadomości o instrukcjach czytania i drukowania w Fortranie. Nie będziemy jednak omawiać wszystkich możliwych parametrów tych instrukcji ograniczając się do minimum pozwalającego w praktyce zrealizować większość typowych operacji.

6.1. Lista sterowania instrukcji `read/write`

Ogólna postać instrukcji czytania `read` i instrukcja drukowania `write` to

`read` (*lista_sterowania*) *lista_i/o*
`write` (*lista_sterowania*) *lista_i/o*

lista_sterowania to lista parametrów sterujących wykonaniem instrukcji
lista_i/o to lista wczytywanych/wypisywanych obiektów

Na liście sterowania mogą pojawiać się oddzielane przecinkami specyfikatory, poniżej omówimy cztery z nich.
Kolejność specyfikatorów na liście jest dowolna, każdy z nich może się pojawić tylko raz.

1) Specyfikator urządzenia (obowiązkowy):

`unit` = *n*

Argument *n* oznacza numer używanego urządzenia. Może to być nieujemne wyrażenie całkowite lub `*` oznaczająca standardowe wejście/wyjście (czyli najczęściej klawiaturę/monitor). Użycie argumentu innego niż gwiazdka omówimy w p. 6.3.
Jeśli specyfikator ten następuje jako pierwszy element listy sterowania, to część „`unit=`” można pominąć.

2) Specyfikator formatu:

`fmt` = *f*

f może być znakiem `*`, stałą tekstową lub nazwą zmiennej tekstowej zawierającej tzw. wzorzec redagowania (jest jeszcze inna możliwość, ale nie będziemy jej tu omawiać). Parametr ten pozwala określić sposób redagowania wczytywanych/wypisywanych znaków. `*` oznacza redagowanie domyślne (tzw. format swobodny). Pozostałe możliwości omówimy w p. 6.2.
Jeśli specyfikator formatu występuje na liście sterowania jako drugi to część „`fmt=`” można pominąć (wtedy należy pominąć także „`unit=`”).

Przykład: instrukcja

```
read(unit=11,fmt=*) x
```

oznacza czytanie zmiennej x w sposób niezredagowany z urządzenia oznaczonego numerem 11. Równoważnie możemy zapisać:

```
read(11,*) x
```
(bo `unit` to pierwszy a `fmt` – drugi argument).

W tym momencie jasne jest już, co oznaczała używana przez nas dotąd postać instrukcji `read` i `write`:

```
read(*,*) ....
write(*,*)....
```

- był to skrócony zapis instrukcji:

```
read(unit=*,fmt=*) ....
write(unit=*,fmt=*) ....
```

oznaczających niezredagowane wczytywanie/wypisywanie informacji z użyciem standardowych urządzeń.

3) Specyfikatory zmiennej przechowującej kod błędu:

`iostat =` *zmiennacałkowita*

Jeśli instrukcja przebiegła pomyślnie, zmienna przyjmuje wartość 0. Po wykryciu błędu operacji wejścia/wyjścia lub wykryciu końca danych przy wczytywaniu zmienna przyjmuje wartość niezerową. W przypadku błędu wartość ta jest większa od zera, a w przypadku wykrycia końca pliku lub rekordu – ujemna, natomiast standard języka nie określa, jaka to ma być konkretnie wartość.

Przykład:

jakieś instrukcje

```
do
  read(*,*,iostat=io) x
  if (io==0) then
     exit
  endif
  write(*,*) 'niepoprawne dane, podaj jeszcze raz'
end do
```

dalsza część programu

Jeśli dane wczytane będą poprawnie, zmienna `io` przyjmie wartość 0 i instrukcja `exit` wyprowadzi nas z pętli.
W przypadku błędu przy podawaniu danych (np. wpisaniu o zamiast 0) wnętrze instrukcji `if` nie zostanie wykonane i po osiągnięciu końca pętli program powróci do próby czytania danych (bez użycia specyfikatora `iostat` w takim przypadku program zakończyłby pracę z komunikatem o błędzie).

4) Specyfikator `advance`

Przyjmuje wartości:

```
advance='yes'
```

lub

```
advance='no'
```

W pierwszym przypadku (jest to przypadek domyślny, jeśli nie użyjemy `advance` jawnie) kolejna instrukcja wejścia/wyjścia będzie wczytywała/wyprowadzała kolejną linię. Jeśli wartość specyfikatora jest równa `'no'`, to transfer danych będzie się odbywał z/do tego samego wiersza.
W przypadku użycia `advance` nie można korzystać z formatu swobodnego.

6.2. Specyfikacja formatu

Porcja danych (pewien ciąg znaków) przetwarzany przez instrukcję `write` lub `read` to tzw. rekord logiczny. Nie musi on koniecznie odpowiadać rekordowi fizycznemu, np. przy wypisywaniu danych jeden rekord logiczny może pojawić się na ekranie jako kilka wierszy (rekordów fizycznych). Specyfikacja formatu podaje, jak interpretować zawartość przetwarzanego rekordu.

Nie będziemy tu omawiać wszystkich szczegółów specyfikacji formatu. W szczególności nie będziemy zajmować się redagowaniem danych podczas wczytywania (z wyjątkiem zmiennych tekstowych). Odpowiada to używaniu `*` jako specyfikacji formatu w instrukcji `read`. Pozwala to użytkownikowi na pewną swobodę podczas podawania danych. Zmuszenie go do umieszczania danych w określonych kolumnach na ogół doprowadzi do błędów wynikających z nieuwagi.
Redagowanie danych ma dla nas większe znaczenie przy wypisywaniu wyników, pozwalając dopasować wygląd wydruku do naszych potrzeb.

Sposób redagowania przetwarzanych danych określamy przez podanie tzw. wzorca redagowania. Ma on postać

(*lista_deskryptorów*)

Wzorzec ten możemy podać:

- w instrukcji `format` (ale tego przypadku nie będziemy omawiać)
- w stałej tekstowej
- jako zawartość zmiennej tekstowej

Na liście deskryptorów podajemy opisy pól oddzielone separatorami. Przecinek służy jedynie do oddzielania poszczególnych elementów listy. Znak / oznacza przejście do następnego rekordu.

Podajmy znaczenie częściej używane oznaczeń pól:

`i`*w* – liczba całkowita w polu o szerokości *w* znaków

`f`*w*`.`*d* – liczba rzeczywista w polu o szerokości *w* znaków z *d* cyframi po kropce

`e`*w*`.`*d* – liczba rzeczywista w zapisie wykładniczym w polu o szerokości *w* znaków z *d* cyframi po kropce

e*w*.*d*e*c* – liczba rzeczywista w zapisie wykładniczym w polu o szerokości *w* znaków z *d* cyframi po kropce, z wykładnikiem zapisanym przy użyciu *c* cyfr
l*w* – wartość logiczna w polu o szerokości *w* znaków
a*w* – łańcuch *w* znaków

Oznaczenie pól może być poprzedzone liczbą naturalną podającą ilość powtórzeń. Nawiasy okrągłe oznaczają powtarzanie części opisu (patrz niżej).
Przy wyprowadzaniu liczba dosuwana jest do prawej krawędzi pola. Jeśli ilość niezbędnych do wyprowadzenia znaków jest większa od szerokości pola, wyprowadzone zostaną gwiazdki.

Przy wyprowadzaniu danych kolejne elementy listy wyjścia wypisywane są w sposób zadany przez kolejne pola wzorca redagowania. Jeśli wyczerpano listę pól we wzorcu a na liście wyjścia pozostały jeszcze jakieś obiekty, wyprowadzany jest nowy rekord. Jeśli w zapisie wzorca nie użyto dodatkowych nawiasów, to jego interpretację zaczyna się od początku, w przeciwnym wypadku powtarzana jest część zawarta w nawiasach.

Przykład:
Wzorzec `(i4,3f10.5)` opisuje pole o szerokości 4 znaków do wyprowadzenia liczby całkowitej i 3 pola o szerokości 10 znaków każde do wyprowadzenia liczb rzeczywistych (z 5 cyframi po kropce) .
Wzorzec `(i4,(e14.5e2))` opisuje pole o szerokości 4 znaków do wyprowadzenia liczby całkowitej i pole o szerokości 14 znaków do wyprowadzenia liczby rzeczywistej w zapisie wykładniczym (z 5 cyframi po kropce i 2 cyframi wykładnika). Ta ostatnia część będzie w razie potrzeby powtarzana.

Wynikiem wykonania programu:

```
program pisz
implicit none

character(10) wzorzec
integer n
real x1,x2,x3

n=3
x1=100.
x2=-0.5
x3=1.e10

write(*,'(i4,3f10.5)') n,x1,x2,x3

write(*,'(i4,(e14.5e3))') n,x1,x2,x3

wzorzec='(/2e14.5)'
write(*,wzorzec) x1,x2,x3

end program pisz
```

będzie utworzenie następującego wydruku:

```
   3 100.00000  -0.50000**********
   3  0.10000E+003
-0.50000E+000
 0.10000E+011

   0.10000E+03  -0.50000E+00

   0.10000E+11
```

W pierwszych dwu instrukcjach `write` wzorzec redagowania podaliśmy w stałej tekstowej, w trzeciej wzorzec został zapisany w zmiennej tekstowej (ten ostatni sposób pozwoliłby nam na modyfikację wzorca w czasie działania programu).
Ponieważ zawartość zmiennej `x3` nie mieści się w polu o zadanej specyfikacji, zostały wypisane gwiazdki.
Druga instrukcja `write` wyprowadziła liczbę całkowitą i liczbę rzeczywistą (wartość zmiennej `x1`). W tym momencie lista pól we wzorcu została wyczerpana a na liście wyjścia pozostały jeszcze `x2` i `x3`. Wyprowadzany jest wobec tego następny rekord a interpretację listy pól zaczyna się powtarzać od nawiasu.
W trzeciej instrukcji `write` nie ma nawiasu oznaczającego część powtarzaną, zatem interpretację wzorca zaczyna się za każdym razem od początku – czyli od znaku / oznaczającego przejście do nowego rekordu – stąd puste linie na wydruku.
Zwróć uwagę na różnicę w działaniu opisów pól `e14.5` i `e14.5e3`.

Przy wczytywaniu bądź wypisywaniu tekstów używa się często określenie pola w postaci `a` dla oznaczenia łańcucha znaków. Przy wczytywaniu jego długość odpowiada długości podanego ciągu znaków, przy wypisywaniu zadeklarowanej długości zmiennej.
Jeśli zmienna `tekst` została zadeklarowana jako zmienna tekstowa o pojemności 10 znaków:

```
character(10) tekst
```

to instrukcja

```
read(*,'(a)') tekst
```

umieści podany z klawiatury ciąg znaków w zmiennej o nazwie `tekst`.
Instrukcja

```
write(*,'(a)') tekst
```

wypisze zaś 10 znaków (tyle zadeklarowano).

We wzorcu redagowania mogą pojawić się jeszcze opisy pól typu: `x` lub *`'tekst'`*. Nie odpowiadają im żadne elementy na liście wyjścia. Specyfikacja `x` opisuje spację (czyli `5x` spowoduje wypisanie 5 znaków odstępu). Drugi opis oznacza wypisanie stałej tekstowej zawartej w apostrofach. Należy zauważyć, że w przypadku zadawania wzorca poprzez stałą tekstową, wewnętrzne apostrofy muszą być podwójne (patrz 8.1):

```
write(*,'(''x= '',f10.5)') x
```

6.3. Użycie plików zewnętrznych

Jak dotąd wszystkie dane wprowadzaliśmy z klawiatury a wyniki wypisywaliśmy na ekran. Przy większej ilości danych/wyników nie jest to wygodna metoda. W takim przypadku wolelibyśmy wczytać dane z pliku (być może zapisanym przez inny program obsługujący

aparaturę pomiarową, być może utworzony przez użytkownika w edytorze tekstu); podobnie wyniki także mogłyby trafić do pliku dyskowego (który potem można analizować).

Aby używać pliku do zapisu/odczytu danych, powinniśmy go wcześniej otworzyć. Służy do tego polecenie `open` :

```
open(unit=nr,file=nazwa)
```

nr definiuje numer urządzenia, *nazwa* to wyrażenie tekstowe (w szczególności stała lub zmienna tekstowa) podające nazwę pliku (poprawną w systemie operacyjnym działającym na komputerze). Zapis „`unit=`” można pominąć, jeśli *nr* podajemy jako pierwszy parametr. Polecenie `open` powoduje otwarcie pliku o podanej nazwie i przypisaniu mu numeru *nr* . W dalszej części programu odwołujemy się do tego pliku przez podanie numeru urządzenia.

W szczególności *nr* możemy podać jako argument specyfikatora `unit` w instrukcji `read` lub `write`. Czytanie/pisanie będzie się wtedy odbywało z/do pliku, któremu przypisaliśmy dany numer urządzenia.

Po wykonaniu wszystkich operacji wejścia/wyjścia z danym plikiem możemy go zamknąć poleceniem

```
close(nr)
```

nr oznacza numer urządzenia przypisany danemu plikowi.

Przykład:

```
program przyklad
implicit none
character(20) nazpl
real x,y
integer n

write(*,*) 'podaj nazwe pliku wynikowego'
read(*,'(a)') nazpl

open(unit=11,file='dane.dat')
open(12,file=nazpl)
     (...)
read(11,*) n
     (...)
write(12,*) x,y

close(11)
close(12)
end program przyklad
```

W powyższym przykładzie użyliśmy pliku z danymi o nazwie `dane.dat` i przypisaliśmy mu numer 11 („`unit=`” moglibyśmy pominąć). Wyniki wypisywane są do pliku o numerze 12, nazwa tego pliku jest podawana przez użytkownika i przechowywana w zmiennej `nazpl`.

Nie omawiamy tu wszystkich szczegółów działania instrukcji `open`. W formie, jaką opisaliśmy, instrukcja ta otworzy istniejący na dysku plik o podanej nazwie lub utworzy nowy, pusty, jeśli plik o tej nazwie jeszcze nie istniał. Oczywiste jest, że plik, z którego czytamy dane, musi wcześniej istnieć. Należy też zauważyć, że wykonanie instrukcji `write` do pliku, który już wcześniej istniał na dysku, skasuje jego poprzednią zawartość.

W przypadku użycia formatu swobodnego podczas czytania

```
read(nr,*)
```

mamy pewną możliwość manewru przy przygotowywaniu pliku z danymi. Otóż każda instrukcja `read` zaczyna wczytywanie kolejnego wiersza. Jeśli ilość danych w wierszu nie wyczerpie listy zmiennych, to czytanie odbywa się z kolejnego wiersza, np. niech x będzie tablicą jednowymiarową, wtedy instrukcja

```
read(11,*) (x(i), i=1,10)
```

będzie czytać z urządzenia (pliku) oznaczonego numerem 11 kolejne elementy tablicy (w sumie 10). W pliku mogą być zapisane w jednym wierszu, w dwu wierszach po 5 elementów, w 3 wierszach po 3 i jednym z 1 elementem, z dodatkowymi pustymi wierszami oddzielającymi – jak nam jest wygodnie.

Kilka porad:
1) Próba czytania lub pisania z urządzenia, którego wcześniej nie otwarto, nie spowoduje błędu. W takim przypadku zostanie otwarty plik o nazwie zależnej od systemu operacyjnego i kompilatora; najczęściej jest to `fort.`*nr,* gdzie *nr* to numer urządzenia.

2) Urządzenia o numerach mniejszych lub równych 10 bywają zarezerwowane do specyficznych celów (np. 5 obsługuje standardowe wejście a 6 standardowe wyjście). Próba otwarcia lub zamknięcia tych urządzeń może dawać nietypowe efekty. Aby uniknąć takich problemów, najlepiej używać numerów urządzeń większych od 10.

3) Oczywiste jest, że w przypadku czytania z pliku poprzedzanie instrukcji `read` instrukcją `write` w rodzaju

```
write(*,*)'podaj dane'
```

nie ma sensu – na tym etapie użytkownik nie może już z tej informacji skorzystać, bo powinien był przygotować plik z danymi przed uruchomieniem programu. Sensowne jest natomiast napisanie osobnej instrukcji podającej użytkownikowi jak przygotować plik z danymi.

7. Podprogramy

7.1. Podprogramy wewnętrzne - wstęp.

Każdy bardziej złożony program możemy podzielić na mniejsze wyodrębnione jednostki (podprogramy). Ich używanie pozwala na uzyskanie bardziej czytelnej struktury programu, uniknięcie zbędnych powtórzeń, ułatwienie testowania programu (podprogramy można testować niezależnie). Oprócz tego podprogramy ułatwiają budowę dużych programów dzięki możliwości „składania" ich z części.

W rozdziale tym zajmiemy się tzw. podprogramami wewnętrznymi. Będą to funkcje oraz podprogramy typu *subroutine* (będziemy je nazywać procedurami). Bliżej omówimy je w następnych podrozdziałach, na razie spróbujmy zobaczyć na prostym przykładzie, jak można ich użyć.

Weźmy prosty program obliczający pole kwadratu o podanym boku:

```
program prosty
implicit none
real x

write(*,*) 'podaj bok kwadratu'
read(*,*) x
write(*,*) 'pole kwadratu= ',x**2
end program prosty
```

i spróbujmy zmodyfikować go dodając dwa podprogramy. Jednym z nich będzie procedura wypisująca na ekranie tekst pożegnalny, drugim funkcja obliczająca kwadrat swojego argumentu.
Nasz program po zmianie wygląda następująco:

```
program mniejprosty
 implicit none
 real x

 write(*,*) 'podaj bok kwadratu'
 read(*,*) x
 write(*,*) 'pole kwadratu= ',kwadrat(x)
 call adios()

contains

 real function kwadrat(y)
 real y
   kwadrat=y*y
 end function kwadrat

 subroutine adios()
 write(*,*) '*************************'
 write(*,*) '* Dziekuje za wspolprace *'
```

```
    write(*,*) '*************************'
    end subroutine adios

    end program mniejprosty
```

Zauważmy, że zdefiniowaną przez nas funkcję `kwadrat` używamy tak samo jak funkcji wewnętrznych Fortranu przez odwołanie się do jej nazwy z podaniem argumentów w nawiasach. Procedurę wywołuje się natomiast poprzez instrukcję `call`, w której podaje się nazwę procedury i argumenty w nawiasach (w naszym przypadku nie potrzeba żadnego argumentu, więc lista jest pusta).

Instrukcje definiujące nasze podprogramy zapisujemy po słowie `contains`, przed `end` kończącym program, między `program` a `contains` zapisane są instrukcje programu głównego.

Definiowanie funkcji zaczynamy instrukcją `function`, w której podajemy nazwę funkcji i listę argumentów. Listę tę nazywamy listą parametrów formalnych, przy ich użyciu definiujemy funkcję. W naszym przypadku jest to zmienna `y`. Zauważmy, że w wywołaniu funkcji użyto zmiennej `x` (jest to tzw. parametr aktualny), jej wartość będzie użyta w miejsce `y` przy obliczaniu funkcji. Ponieważ `y` jest typu `real`, więc parametr aktualny przy wywołaniu funkcji musi być tego samego typu.

Zauważmy, że w treści funkcji pominięto dyrektywę `implicit none`, a to dlatego, że zostaje ona „odziedziczona” z programu głównego. Funkcja zwraca wartość określonego typu, zatem musimy określić typ wyniku funkcji. W naszym przypadku informuje o tym deklaracja typu `real` przed słowem `function`. Alternatywnie moglibyśmy ją tam pominąć, natomiast dodać nazwę funkcji do deklaracji typów:

```
    function kwadrat(y)
    real kwadrat,y
```

Instrukcja

```
    kwadrat =
```

przypisuje wartość wyniku funkcji `kwadrat`. Instrukcja taka musi pojawić się w definicji funkcji, inaczej wynik będzie nieokreślony.

Definicję funkcji kończy instrukcja `end function`. Po jej wykonaniu do programu wywołującego przekazana zostaje wartość wyniku funkcji.

Procedura `adios` służy do wypisania na ekranie tekstu. Początek procedury sygnalizuje słowo `subroutine`, po którym podaje się jej nazwę i w nawiasach listę parametrów formalnych (argumentów). W naszym przypadku nie potrzebowaliśmy żadnego argumenty, więc lista jest pusta.

Ponieważ procedura nie zwraca wyniku, nie można jej przypisać typu.

Procedurę kończy instrukcja `end subroutine`, której wykonanie przenosi sterowanie do programu, który wywołał procedurę.

Skomplikujmy jeszcze trochę nasz program dodając procedurę `czytaj`, która będzie ponawiała czytanie długości boku kwadratu, dopóki nie zostanie podana wartość poprawna (tzn. nieujemna). Oto nasz program:

```
      program corazmniejprosty
       implicit none
       real x

       call czytaj(x)
       write(*,*) 'pole kwadratu= ',kwadrat(x)
       call adios()

      contains

       real function kwadrat(y)
       real y
         kwadrat=y*y
       end function kwadrat

       subroutine adios()
         write(*,*) '**************************'
         write(*,*) '* Dziekuje za wspolprace *'
         write(*,*) '**************************'
       end subroutine adios

       subroutine czytaj(y)
         real y

         do
           write(*,*) 'podaj bok kwadratu'
           read(*,*) y
           if (y>=0) then
             exit
           endif
           write(*,*) 'zla wartosc'
         enddo
       end subroutine czytaj

end program corazmniejprosty
```

Zauważmy, że kolejność umieszczenia podprogramów po `contains` jest dowolna i nie ma żadnego związku z kolejnością ich wywoływania.

Parametr formalny naszej procedury nazywa się `y`. Podczas pracy procedury zmiennej tej zostaje nadana wartość, która po przekazaniu sterowania do programu głównego zostaje zwrócona jako wartość zmiennej użytej jako parametr aktualny przy wywołaniu procedury (w naszym przypadku `x`). W przypadku procedur modyfikacja przez nie wartości swojego argumentu nie jest niczym niezwykłym, natomiast w przypadku funkcji byłoby to nieintuicyjne.

Zwróćmy uwagę, że następujące wywołanie funkcji `kwadrat` jest poprawne

```
a = kwadrat(2.0)
```

(argumentem może być stała odpowiedniego typu), natomiast w przypadku procedury `czytaj` użycie stałej

```
call czytaj(2.0)
```

spowoduje błąd przy wykonaniu programu (bo instrukcja `read` nie może wczytać wartośći do stałej). Bliżej o tym problemie powiemy w p. 7.5.

7.2. Podprogramy wewnętrzne – cd.

Zacznijmy systematyzować informacje z poprzedniego podrozdziału.

Podprogramy wewnętrzne zapisuje się dodając po ostatniej wykonywalnej instrukcji programu między słowem `contains` a kończącym program `end program`. Jeżeli w programie użyto `contains`, dalej musi pojawić się zapis przynajmniej jednego podprogramu.

Struktura programu wygląda zatem następująco:

`program` *nazwa*

instrukcje programu

`contains`

podprogramy

`end program` *nazwa*

Z kolei zapis podprogramów jest następujący (pokazano minimalne specyfikacje, bardziej rozbudowane pojawią się później):
- podprogram typu funkcji

typ `function` *nazwa_funkcji*(*lista_argumentów*)

treść funkcji

`end function` *nazwa_funkcji*

- procedura

`subroutine` *nazwa_proc*(*lista_argumentów*)

treść podprogramu

`end subroutine` *nazwa_proc*

Nazwa_funkcji lub *nazwa_proc* to nazwa nadana podprogramowi; po tej nazwie jest on rozpoznawany.

W kończącej podprogram wewnętrzny instrukcji `end` nie można opuścić słowa `function` lub `subroutine` (natomiast można w tym miejscu pominąć nazwę).

Lista_argumentów zawiera oddzielone przecinkami nazwy argumentów podprogramu (tzw. parametry formalne). Przy użyciu tych nazw programuje się instrukcje podprogramu. Przy wywołaniu podprogramu natomiast podaje się tzw. listę parametrów aktualnych – czyli konkretne wartości argumentów, dla których mają być wykonane instrukcje. Podczas wykonywania instrukcji podprogramu pod parametr formalny (czyli użyty przy definiowaniu podprogramu) podstawia się odpowiadający mu parametr aktualny. Liczba i typ argumentów muszą być zgodne.

Jeśli podprogram nie wymaga argumentów, *lista_argumentów* może być pusta.

Musimy teraz wyjaśnić sprawę zasięgu deklaracji użytych w programie i podprogramach. Jak widać ze schematu powyżej, podprogramy wewnętrzne są osadzone w programie głównym, który pełni dla nich rolę „gospodarza" (*host*).

Użycie w programie głównym dyrektywy `implicit none` oznacza, że obowiązuje ona także w podprogramach (można ją tam zatem opuścić).

Podobnie dziedziczone są deklaracje zmiennych. Zadeklarowanie w programie głównym (*hoście*) zmiennej oznacza, że będzie ona dostępna w podprogramach (jeśli nie „nadpiszą" tej deklaracji) – jest to więc zmienna dostępna poprzez tzw. *host association*. Jeśli jednak podprogram jawnie zadeklaruje zmienną o tej samej nazwie, to będzie to już zmienna *lokalna*, która nie ma nic wspólnego ze zmienną o tej samej nawie użytej w programie głównym.
Dzięki dyrektywie `implicit none` wymuszającej jawne deklaracje mamy zatem czystą sytuację: jeśli podprogram używa jakiejś zmiennej to albo została w nim zadeklarowana (i jest to wtedy jego zmienna lokalna), albo jest to zmienna zadeklarowana w programie głównym i udostępniona stamtąd przez *host association*.

Przykład:

```
program okaz
   implicit none
   real x,y,z
   real a,b
      ....
   z = dziwna(y)
      .....

contains

   function dziwna(x)
      real dziwna,x,y
      integer z,b
          ...
```

```
    end function dziwna
end program okaz
```

W powyższym przykładzie funkcja `dziwna` deklaruje zmienne o nazwach `x`, `y`, `z` i `b` i są to jej zmienne lokalne, różne od użytych w programie głównym zmiennych o tej samej nazwie. Zmienna `y` z programu głównego w podprogramie dostępna jest jako `x`, ponieważ została przekazana w wywołaniu funkcji (*argument asscociation*). Ponieważ w funkcji nie zadeklarowano zmiennej `a`, to użycie tej zmiennej w funkcji oznacza odwołanie się do zmiennej `a` dostępnej z programu głównego przez *host association*.
Dla porządku powiedzmy jeszcze, co stałoby się, gdyby w deklaracjach użytych w funkcji `dziwna` nie zadeklarowano zmiennej y. Wtedy w funkcji `dziwna` dostępna byłaby zmienna `y` z programu głównego (poprzez *host association*). Równocześnie jednak byłaby dostępna jako zmienna `x` (w wyniku wywołania funkcji, przez *argument association*). W takim przypadku wewnątrz podprogramu wolno się odwoływać do tej zmiennej jedynie poprzez parametr formalny (czyli w powyższym przypadku jako `x`).

Nazwy podprogramów w obrębie *hosta* (programu głównego) są dostępne wszędzie – zatem mogą się nawzajem wywoływać.

Podprogramy wewnętrzne same nie mogą zawierać podprogramów.

7.3. Podprogramy typu `function`

Podprogram typu funkcji (`function`) realizują w Fortranie obliczanie funkcji definiowanej przez użytkownika. Funkcja zwraca jako efekt swojego działania pewną wartość, wartość jest jednego z typów używanych w Fortranie.

Podprogram taki rozpoczyna się od instrukcji `function` w postaci

typ `function` *nazwa_fun* (*lista_argumentów*)

gdzie *typ* to określenie typu wartości funkcji
nazwa_fun to nazwa definiowanej funkcji
lista_argumentów zawiera oddzielone przecinkami argumenty funkcji, tzw. parametry formalne, może też być pusta, jeśli funkcja nie wymaga podania argumentów

Określenie typu funkcji w instrukcji `function` można pominąć, wtedy do określenia typu trzeba użyć odpowiedniej deklaracji.

Po instrukcji `function` następuje ciąg instrukcji służących do znalezienia wartości funkcji. Wśród nich przynajmniej raz musi znaleźć się instrukcja przypisania:

nazwa_fun = *wyrażenie*

nadająca funkcji wartość zadaną wyrażeniem po prawej stronie znaku =

Na końcu podprogramu wewnętrznego definiującego funkcję musi znaleźć się instrukcja `end function`. Jej wykonanie kończy pracę podprogramu i przenosi sterowanie do

programu wywołującego, zwracając wartość funkcji nadaną ostatnio wykonaną instrukcją przypisania typu *nazwa_fun* = ...

Zdefiniowaną w postaci podprogramu funkcję używa się tak samo, jak funkcje standardowe Fortranu przez podanie nazwy funkcji i w nawiasach listy jej argumentów (są to tzw. parametry aktualne).
Liczba i typ parametrów aktualnych (przy wywołaniu funkcji) musi być zgodny z liczbą i typem parametrów formalnych (definiujących funkcję). Mogą to być zmienne, stałe lub bardziej złożone wyrażenia odpowiedniego typu. Nienaturalne (choć dopuszczalne i wykorzystywane) jest modyfikowanie przez funkcję swoich argumentów. W takim przypadku obowiązują ograniczenia opisane w 7.5

Przykład: program wywołujący funkcję obliczającą sumę kwadratów kolejnych liczb naturalnych od 1 do zadanej liczby.

```
program sumujkw
  implicit none
  integer n

  write(*,*)'podaj n'
  read(*,*) n
  write(*,*)'suma kwadratow od 1 do ',n,' wynosi ',sumakw(n)

contains

  function sumakw(m)

    integer sumakw, i, m

    sumakw=0
      do i=1,m
        sumakw=sumakw+i*i
      end do

   end function sumakw
end program sumujkw
```

Zwróćmy uwagę na parę szczegółów:
- Typ funkcji określiliśmy przy pomocy deklaracji w obrębie podprogramu; alternatywnie moglibyśmy zapisać:
  `integer function sumakw(m)`
- W programie głównym nie określaliśmy typu funkcji, ponieważ jest on dla programu znany, gdyż jest to jego funkcja wewnętrzna
- W definicji funkcji użyliśmy zmiennej `m` do oznaczenia granicy sumowania (parametr formalny). W momencie wywołania funkcji jako parametr aktualny użyta została zmienna `n`. Jej wartość została przekazana do funkcji i użyta tam, gdzie w przepisie występowała nazwa `m`.
- Zmienna `i` została zadeklarowana w podprogramie, zatem jest to jego zmienna lokalna. W związku z tym równie dobrze moglibyśmy zapisać nasz program następująco:

```
program sumujkw
  implicit none
  integer n,i

  write(*,*) 'podaj n'
  read(*,*) n
  i=sumakw(n)
  write(*,*) 'suma kwadratow od 1 do ',n,' wynosi ',i
contains
      ....
```

W programie głównym i podprogramie zmienna o nazwie `i` jest wykorzystana do innych celów `i` nie ma żadnego związku pomiędzy nimi.
W opisanym przykładzie wywołanie:

```
n = sumakw(20)
```

jest poprawne (przypisze zmiennej `n` wartość sumy kwadratów liczb od 1 do 20), natomiast

```
n = sumakw(20.0)
```

już nie (niezgodność typów parametrów formalnych i aktualnych)

7.4. Podprogramy typu `subroutine`

Innym typem podprogramu jest procedura (*subroutine*). Opisuje ona ciąg czynności do wykonania, natomiast w przeciwieństwie do funkcji nie ma przypisanej wartości.

Podprogram definiujący procedurę zaczyna się od instrukcji `subroutine`:

`subroutine` *nazwa_sub*(*lista_argumentów*)

nazwa_sub to nazwa procedury a znaczenie *listy argumentów* jest takie samo, jak w przypadku funkcji.

Należy zwrócić uwagę, że procedurze `subroutine` nie można przypisać typu, nielegalna jest też instrukcja podstawienia wartości pod *nazwa_sub*.

Procedurę wywołuje się instrukcją `call`:

`call` *nazwa_sub*(*lista_parametrów_aktualnych*)

Zakończenie pracy procedury i przekazanie sterowania do programu wywołującego następuje po wykonaniu instrukcji `end subroutine` umieszczonej na końcu podprogramu.

Zasady przekazywania argumentów do procedury są takie same, jak w przypadku funkcji. Warto jednak zauważyć, że w przeciwieństwie do funkcji, modyfikowanie przez procedurę wartości swoich argumentów jest naturalne – w ten sposób do programu wywołującego przekazywane są wyniki działania procedury.

Zobaczymy przykład procedury zamieniającej wartości swoich argumentów:

```
program zamien
  implicit none
  real c,d

  c=10.0
  d=5.
  call swap(c,d)

  write(*,*)'c =',c,' d=',d

contains

  subroutine swap(a,b)
  real a,b,tmp

  tmp=b
  b=a
  a=tmp

  end subroutine swap
end program zamien
```

Zauważ, że do zamiany wartości zmiennych `a` i `b` potrzebujemy zmiennej tymczasowej, która przez chwilę przechowa wartość jednej ze zmiennych (do wymiany zawartości dwu pełnych szklanek potrzebujemy trzeciej pustej).

Po wykonaniu programu stwierdzimy, że wypisana wartość zmiennej `c` będzie równa 5.0 a wartość `d` będzie 10.0 . Jak widać, nazwy zmiennych będących parametrami aktualnymi nie mają nic wspólnego z nazwami parametrów formalnych (bo te ostatnie są jawnie zadeklarowane, czyli lokalne).

7.5. Przekazywanie danych do podprogramów. Atrybut `intent`.

Powróćmy teraz do dotychczas pomijanego szczegółu: jak przekazywane są argumenty.

Jeśli parametrem aktualnym jest zmienna (w tym tablica, lub jej wycinek), podprogramowi udostępniany jest jej adres w pamięci. Oznacza to, że jeśli w wyniku działania podprogramu parametr formalny odpowiadający temu parametrowi aktualnemu zmieni wartość, to nowa wartość będzie dostępna w programie wywołującym jako wartość odpowiedniej zmiennej. Jak wspomnieliśmy wyżej jest to naturalne dla procedur, natomiast modyfikacja argumentu przez funkcję jest nieintuicyjna i może doprowadzić do efektów ubocznych zatem powinna być stosowana jedynie w naprawdę niezbędnych przypadkach.

Jeżeli parametrem aktualnym jest stała lub wyrażenie, do podprogramu przekazywana są ich wartości. Jeśli w takim przypadku w trakcie działania podprogramu nastąpi zmiana

odpowiedniego parametru formalnego, to nie ma możliwości przekazania jej do programu wywołującego (bo nie można zapisać tej nowej wartości w stałej lub wyrażeniu).

Np. poniższe wywołanie procedury `swap` zdefiniowanej w podrozdziale poprzednim jest niepoprawne:

```
call swap(10.0,5.0)
```

ponieważ procedura modyfikuje wartości argumentów.

Zatem, jeśli podprogram modyfikuje parametry formalne, to parametry aktualne przy wywołaniu podprogramu muszą być zmiennymi. Nieprzestrzeganie tej zasady prowadzi do błędnego działania programu.

Potencjalne niebezpieczeństwo błędów można zredukować, jeśli w podprogramie przy deklaracji zmiennych będących parametrami formalnymi określimy ich atrybut nazwany `intent`, ponieważ kompilator może wtedy sprawdzić poprawność przekazywanych argumentów.

Odpowiednia deklaracja ma postać:

typ, `intent(`*specint*`) ::` *lista_nazw_zmiennych*

lista_nazw_zmiennych zawiera oddzielone przecinkami nazwy zmiennych, wszystkie muszą być parametrami formalnymi.
specint przybiera jedną z trzech wartości: `in`, `out` lub `inout` .

`intent(in)` oznacza, że dana parametr formalny służy tylko do przekazania danych do podprogramu. Nie może być w podprogramie modyfikowany (kompilator wyświetli w takim przypadku komunikat o błędzie).

`intent(out)` oznacza, że dany parametr formalny służy tylko do przekazania danych z podprogramu do programu wywołującego. W trakcie wykonania podprogramu musi mu zatem zostać nadana wartość. Odpowiadający mu parametr aktualny musi być zmienną a nie stałą czy wyrażeniem.

`intent(inout)` – parametr służy do przekazania danych do i z podprogramu, zatem odpowiedni parametr aktualny musi być zmienną.

Przykład:

```
program lezem_i_kwiczem
implicit none
real x
                  ......
   call klapa(x,2.0)
                   .....
contains
      subroutine klapa(a,b)
         real, intent(in) :: a
         real, intent(out) :: b
```

```
            ....
        a=a+1
              ....
    end subroutine klapa
end program lezem_i_kwiczem
```

Powyższy program nie zostanie skompilowany, ponieważ pierwszy argument nie może być modyfikowany a drugi parametr aktualny nie może być stałą.

7.6. Tablice jako parametry formalne

Jeśli parametrem formalnym funkcji lub procedury jest zwykła zmienna, to na liście parametrów aktualnych możemy użyć w jej miejscu zmiennej lub pojedynczego elementu tablicy (lub stałej albo wyrażenia jeśli wartość tego argumentu nie jest modyfikowana). Parametrem formalnym może być też tablica. W tym przypadku na liście argumentów podajemy tylko jej nazwę, natomiast wymiary musimy zadeklarować w obrębie podprogramu.

Jeśli rozmiar tej tablicy jest stały i znany już na etapie pisania programu możemy użyć deklaracji statycznej w sposób opisany w p. 5.1

W przeciwnym przypadku najprościej jest użyć specyfikacji w postaci:

typ nazwatablicy`(:, :, ....)`

Podajemy tyle oddzielonych przecinkami dwukropków, ile indeksów ma dana tablica.

Parametr formalny zadeklarowany w ten sposób przyjmuje kształt parametru aktualnego użytego w wywołaniu podprogramu, tzn. każdy indeks przyjmuje tyle wartości, ile wynika z rozmiaru przesłanej do podprogramu tablicy. Jest to tzw. *assumed shape array*.

Przykład:

```
program tabele
real  x(3,3), z(5,5)
real, allocatable :: y(:,:,:)
          ....
allocate(y(10,5,3))
          ....
call tab(x,y,z(1:5:2,1:5:2))
          ....
contains

   subroutine tab(a,b,c)
     real a(3,3), b(:,:,:), z(:,:)
          ....
   end subroutine tab
end program tabele
```

W podprogramie tablica `a` deklarowana jest statycznie jako tablica 3×3. Ponieważ taki sam jest kształt i rozmiar odpowiedniego parametru aktualnego (tablica `x` w programie głównym) odpowiedniość elementów obu tablic jest oczywista.

Deklaracja tablicy `b` określa, że ma trzy indeksy. Przy wywołaniu programu przyjmie kształt tablicy `y`, czyli indeksy będą się zmieniać odpowiednio od 1 do 10, od 1 do 5 i od 1 do 3.

Deklaracja tablicy `c` mówi, że jest ona dwuwymiarowa. Jako parametr aktualny przesłano dwuwymiarową tablicę, będącą wycinkiem tablicy `z` (konkretnie tablicę 3×3 składającą się z elementów z(1,1), z(3,1), z(5,1), z(1,3), z(3,3), z(5,3), z(1,5), z(3,5) i z(5,5)). W podprogramie tablica `c` przyjmie kształt 3×3, ale jej elementy są inaczej indeksowane: c(1,1), c(2,1), c(3,1), c(1,2), c(2,2), c(3,2), c(1,3), c(2,3), c(3,3). Zwróćmy uwagę, że odpowiadające sobie elementy parametru formalnego i aktualnego określamy nie na podstawie indeksów a położenia w tablicy (np. element z 3 kolumny 1 wiersza).

7.7. Rekurencyjne wywoływanie podprogramów.

Fortran 90 pozwala na rekurencyjne wywoływania podprogramów – zarówno jawne (podprogram wywołuje sam siebie) jak i niejawne (podprogram A wywołuje podprogram B, który wywołuje podprogram A).

Jeżeli podprogram ma być wywoływany rekurencyjnie, musimy poinformować o tym kompilator umieszczając słowo `recursive` przed `function` lub `subroutine` w deklaracji podprogramu, np.:

```
recursive subroutine dzielprzedzial(a,b)
```

W przypadku funkcji istnieje dodatkowa komplikacja: w instrukcjach definiujących funkcję musi pojawić się przypisanie:

nazwa_funkcji = ...

Jeśli funkcja miałaby odwołać się do samej siebie, to *nazwa_funkcji* pojawiłaby się po prawej stronie znaku = powyższej instrukcji. Jest to nielegalne; w takim przypadku musimy wprowadzić osobną nazwę dla wyniku funkcji. Robimy to określając opcję `result` :

typ `recursive function` *nazwa_funkcji*(*lista_arg*) `result`(*nazwa_wyniku*)

lub

`recursive` *typ* `function` *nazwa_funkcji*(*lista_arg*) `result`(*nazwa_wyniku*)

(jak widać kolejność określenia typu i `recursive` nie ma znaczenia)

Jeżeli używamy opcji `result`, to:
- do nadania wartości funkcji używamy przypisania
  nazwa_wyniku = ...
- *nazwa_funkcji* używana jest do rekurencyjnych wywołań funkcji
- jeśli typ wyniku funkcji deklarujemy w osobnej deklaracji, to używamy tam *nazwy_wyniku*, nie *nazwy_funkcji*

- program wywołujący naszą funkcję używa *nazwy_funkcji*; jako wynik otrzyma wartość nadaną w podprogramie instrukcją

nazwa_wyniku = ...

Opcja `result` może być określona dla każdej funkcji, ale wymagana jest jedynie dla funkcji z jawną rekurencją.

Zobaczmy na przykładzie, jak może wyglądać rekurencyjne wywoływanie funkcji. Napiszemy w tym celu podprogram obliczający silnię z rekurencyjnej zależności:

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

Uwaga: przykład służy jedynie do zademonstrowania rekurencji, obliczanie silni jej nie wymaga (patrz p. 4.4). W ogólności, jeśli istnieje prosta nierekurencyjna metoda rozwiązania problemu, należy jej użyć. Rekurencję stosujemy w przypadkach, kiedy nierekurencyjne rozwiązanie jest zbyt skomplikowane.

```
program silaczka
  implicit none
  integer n

  write(*,*) 'podaj n'
  read(*,*) n

  write(*,*) n,'! = ',silnia(n)

contains
  recursive function silnia(n) result(sil)
  integer sil,n

  if (n==0) then
    sil = 1
  else
    sil = n * silnia(n-1)
  endif
  end function silnia
end program silaczka
```

Nazwa funkcji to `silnia` i ta nazwa używana jest do jej wywołań. Natomiast nazwa wyniku używana w podprogramie to `sil` i to ta nazwa występuje w deklaracji typu oraz przy nadawaniu funkcji wartości.
Zwróćmy uwagę na fakt, iż przy rekurencyjnych podprogramach należy zadbać o warunek powodujący przerwanie łańcucha rekurencyjnych wywołań (w naszym przypadku jest to nadanie funkcji wartości 1, gdy argument jest równy 0). W przeciwnym wypadku otrzymamy program, który będzie wywoływał rekurencyjnie podprogram do wyczerpania zasobów komputera. (Zastanów się, co stanie się w przypadku podania ujemnego `n`)

8. Operacje na tekstach

8.1. Stałe, zmienne znakowe, wycinki

W poprzednich rozdziałach spotykaliśmy się ze stałymi i zmiennymi tekstowymi. W bieżącym rozdziale uzupełnimy te informacje.

Stałą tekstową jest ciąg znaków ujęty w apostrofy; apostrofy te nie są częścią stałej. Jeśli stała znakowa ma zawierać apostrof, kodujemy go przy pomocy dwu znaków apostrofu.

Przykłady stałych tekstowych:

```
'Ala ma kota'
'Kot ma pchły'
'x''y'
```

Ta ostatnia stała to x'y . W stałych tekstowych możemy użyć znaków, które nie należą do alfabetu Fortranu.

Zmienne znakowe przechowują ciągi znaków. Deklaracja zmiennych znakowych ma postać:

`character(`*długość*`)` *lista_nazw_zmiennych*

Jeśli nie podano długości, to zmienna jest jednoznakowa:

```
character odp
character(10) nazwa1
character(50) dluganaz
```

długości zmiennych `odp`, `nazwa1` i `dluganaz` to odpowiednio 1,10 i 50 znaków.

Długość zmiennej tekstowej będącej parametrem formalnym podprogramu można zadeklarować z użyciem znaku *.

```
subroutine okaz1(nazwa)
character(*) nazwa
```

Zmienna przyjmie wtedy długość parametru aktualnego użytego w wywołaniu podprogramu.

Możliwe jest także deklarowanie tablic typu znakowego, np. deklaracje

```
character(20) imie(100)
character(30) nazwisko(100)
```

mówią, że tablica `imie` przechowuje 100 tekstów o długości 20 znaków, a tablica `nazwisko` 100 tekstów po 30 znaków.

Fortran umożliwia odwoływanie się do wycinków zmiennych znakowych przez podanie nazwy zmiennej i zakresu pozycji tworzących wycinek:

nazwa (*zp* : *zk*)

gdzie *nazwa* to nazwa zmiennej, a *zp* i *zk* są wyrażeniami całkowitymi. *zp* to pozycja lewego końca wycinka (jeśli nie jest podana, to *zp* jest równe 1), *zk* to pozycja prawego końca wycinka (jeśli nie jest podana, to oznacza wycinek do końca zmiennej).

Przykład: dla zmiennej:

```
abecadlo='abcdefgh'
```

wycinki mają wartości:

`abecadlo(3:4)` to 'cd'
`abecadlo(5:5)` to 'e'
`abecadlo(:4)` to 'abcd'
`abecadlo(6:)` to 'fgh'
`abecadlo(:)` to 'abcdefgh'

Wycinki tablic znakowych podajemy w postaci:

nazwa`(`*ind_el*`)(`*zp*`:`*zk*`)`

(w pierwszej parze nawiasów podajemy indeks(y) elementu tablicy; w drugiej zakres wycinka), np.

```
nazwisko(10)(1:15)
```

to wycinek od 1 do 15 znaku 10-go elementu tablicy `nazwisko`.

8.2. Operacje na tekstach

Jak już wspomnieliśmy, przy wprowadzaniu lub wyprowadzaniu zmiennych tekstowych używamy specyfikatora pola `a` , np.

```
character(10) nazwa

read(*,'(a)') nazwa
```

Instrukcja przypisania dla wyrażeń znakowych ma postać

nazwa = *wyrażenie*

gdzie *nazwa* to zmienna tekstowa, element tablicy tekstowej lub wycinek

Wyrażenie znakowe po prawej stronie instrukcji przypisania konstruuje się ze stałych, zmiennych, elementów tablic, wycinków i wywołań funkcji znakowych z ewentualnym wykorzystaniem nawiasów i operatora konkatenacji.

Operator konkatenacji zapisywany jako `//` (dwa slash'e) łączy dwa łańcuchy znakowe, między którymi jest zapisany, np. wynikiem działania

```
'abc'//'efg'
```

jest łańcuch znaków `'abcdef'`.

Pożyteczną funkcją operującą na zmiennych znakowych w Fortranie jest funkcja `index`. Funkcja ta wywołana w postaci `index(w1,w2)`, gdzie `w1` i `w2` to łańcuchy znaków, zwraca wartość całkowitą będącą pozycją początku pierwszego wystąpienia łańcucha `w2` w łańcuchu `w1` (albo 0, jeśli nie występuje).
Np. po wykonaniu fragmentu programu:

```
character(20) zdanie
integer i1,i2

zdanie='As to Ali pies'

i1=index(zdanie,'kot')
i2=index(zdanie,'pies')
```

`i1` będzie równe 0 (bo tekst `'kot'` nie występuje w łańcuchu znaków `'As to Ali pies'`), a `i2` będzie równe 11 (bo początek łańcucha `'pies'` znajduje się na 11 pozycji przeszukiwanego tekstu).

Do porównywania wartości znakowych można wykorzystać operatory relacji znane z rozdziału 3.1, z tym, że z wyjątkiem operatorów == i /= wynik zależy od uporządkowania znaków w konkretnej implementacji. Na przykład standardowo litery alfabetu łacińskiego uporządkowane są alfabetycznie a cyfry od 0 do 9, ale nie wiadomo, czy litery mają występować przed cyframi, czy po (ten drugi przypadek ma miejsce dla kodów ASCII). W szczególności należy pamiętać, że polskie litery na pewno nie będą w uszeregowaniu zbioru znaków na oczekiwanych miejscach, zatem przy sortowaniu alfabetycznym tekstów zawierających polskie znaki nie można wykorzystać operatorów języka Fortran (trzeba napisać własne funkcje do porównywania tekstów).

8.3. Zmienne znakowe jako pliki wewnętrzne

Warto wspomnieć o zastosowaniu zmiennych tekstowych jako tzw. plików wewnętrznych przy operacjach czytania i pisania. Przypuśćmy, iż nasz program ma przeczytać wartości dwu zmiennych całkowitych `nx` i `ny`. Wartości te podane są w jednej linii pliku z danymi i poprzedzone są opisem, tzn. odpowiedni wiersz pliku ma postać np.

```
nx=   23          ny=     100
```

Niestety dane dostajemy od kogoś i nie mamy wpływu na sposób ich zapisu. Wiemy jedynie, że wartości zmiennych poprzedzone są tekstami ‘nx=’ oraz ‘ny=’, ale nie znamy ich położenia w wierszu, zatem nasza linia pliku równie dobrze mogłaby wyglądać następująco:

```
nx=23 ny= 100
```

Widzimy zatem, że nie jesteśmy w stanie przewidzieć, w których kolumnach znajdują się potrzebne wartości. Nie pozostaje nam zatem nic innego, jak wczytać cały wiersz jako ciąg

znaków (zmienna tekstowa), przeanalizować go i następnie próbować wydostać potrzebne dane z odpowiednich kolumn.

Pomoże nam w tym możliwość użycia pliku wewnętrznego. Otóż jako urządzenie wejścia/wyjścia (specyfikator `unit`) można podać nazwę zmiennej znakowej, z której/do której należy czytać/pisać dane. Należy mieć na uwadze fakt, iż nie można wtedy użyć formatu swobodnego (czyli gwiazdki jako specyfikacji formatu).
Jeżeli zatem stwierdziliśmy, że liczba całkowita, którą chcemy wczytać znajduje się w kolumnach od 15 do 20 zmiennej tekstowej `linia`, to możemy przeczytać wartość zmiennej całkowitej (nazwijmy ją `n`) instrukcją

```
read(linia(15:20),'(i6)') n
```

W specyfikacji formatu pojawia się `i6`, bo pole, z którego czytamy ma 6 znaków szerokości.
Zauważmy jednak, że w praktyce nie będziemy mogli skorzystać ze stałych, tylko ze zmiennych: stwierdzimy np, że dane zapisane są w kolumnach od `i` do `j`, czyli w polu o szerokości `szp = j - i + 1` znaków. Jak wobec tego podać tę wartość w specyfikacji formatu? – wykorzystując jako plik wewnętrzny inną zmienną tekstową, do której wpiszemy szerokość pola:

```
form = '(i )'
write(form(3:3),'(i1)') szp
```

W odpowiedni znak (trzeci) zmiennej tekstowej `form` wpisaliśmy wartość szerokości pola (zauważmy jednak, że milcząco założyliśmy, że szerokość ta jest mniejsza od 10). Możemy teraz wykorzystać zmienną `form` w instrukcji czytania ze zmiennej `linia` :

```
read(linia(i:j),form) n
```

Pozostaje jeszcze wyjaśnić, jak ustalić pola, w których mamy szukać liczby – oczywiście ustalając przy pomocy funkcji `index` położenie tekstu, który poprzedza naszą liczbę.
Zobaczmy wobec tego fragment programu czytającego nasze `nx` i `ny` z pliku dane.dat z wykorzystaniem zmiennej `linia`.

```
program przyklad
implicit none
integer, parameter :: maxl = 80     ! to wyjasnimy nizej
character(maxl) linia
character(5) form
integer inx,iny,szer,nx,ny

open(11,file='dane.dat')
read(11,'(a)') linia

inx=index(linia,'nx=')
iny=index(linia,'ny=')

form='(i  )'

szer=(iny-1)-(inx+3)+1
```

```
call pisz_szer(form,szer)
read(linia(inx+3:iny-1),form) nx

szer=maxl-(iny+3)+1
call pisz_szer(form,szer)
read(linia(iny+3:maxl),form) ny

.....

contains

subroutine pisz_szer(form,sz)
character(*) form
integer sz

if (sz < 10) then
    write(form(3:3),'(i1)') sz
else
    write(form(3:4),'(i2)') sz
endif
end subroutine pisz_szer

end program
```

A teraz wyjaśnienia:
Atrybut `parameter` przypisany jakiejś zmiennej oznacza, iż będzie to stała, której już nie będzie można zmienić w programie (ale można ją wykorzystać np. w deklaracjach innych zmiennych). Wartość stałej nadaje się w instrukcji przypisania po `::`
My użyliśmy tej konstrukcji do określenia stałej `maxl` podającej maksymalną długość wiersza.

Procedurę `pisz_szer` użyliśmy do wpisania odpowiedniej szerokości pola do specyfikacji formatu przechowywanej w zmiennej `form` (jak widać będzie działać poprawnie dla pól o szerokości < 100 znaków)
Gwiazdka w specyfikacji długości zmiennej znakowej w podprogramie oznacza, iż zmienna będzie miała długość odpowiadającą parametrowi aktualnemu w wywołaniu podprogramu.

9. Struktury

9.1. Najpierw przykład...

Zauważmy, iż niejednokrotnie mamy do czynienia ze zmiennymi, które są jakoś ze sobą powiązane. Na przykład do zdefiniowania okręgu potrzebujemy podać współrzędne jego środka i promień – mamy zatem odpowiednie zmienne:

```
real x0,y0,r
```

Pisząc program w ten sposób musimy pamiętać o związku logicznym, jaki zachodzi między tymi zmiennymi.

Fortran pozwala w takim przypadku zdefiniować nowy, określony przez użytkownika typ zmiennej – strukturę, której składowymi są nasze zmienne. W naszym przypadku definicja taka może wyglądać następująco

```
type circle
     real x0,y0
     real r
end type circle
```

a wykorzystanie naszego nowego typu w deklaracji zmiennych:

```
type (circle) okrag1, okrag2
```

Powyższe deklaracje oznaczają, że

wprowadziliśmy nowy typ zmiennej nazywając go `circle`
zmienne typu circle zawierają trzy składowe rzeczywiste nazwane `x0,y0,r`
w drugiej deklaracji powiedzieliśmy, że zmienne `okrag1` i `okrag2` są typu `circle`

Zobaczmy, jak możemy wykorzystać tzw. konstruktor struktury do nadania wartości zmiennej typu `circle`:

```
okrag1 = circle(1.0,0.0,5.3)
```

Nadaliśmy składowym `x0,y0` oraz `r` struktury okrag1 wartości odpowiednio 1.0, 0.0 oraz 5.3.

Mamy dostęp do poszczególnych składowych zmiennej:

```
okrag2%x0 = 2.0
okrag2%y0 = okrag2%x0
okrag2%r = 10.0
```

ale możemy też odwoływać się do niej jako całości:

```
okrag2 = okrag1
write(*,*) okrag2
```

9.2. A teraz teoria

Definicja nowego typu (struktury) ma postać

`type` *nazwa*

lista deklaracji zmiennych

`end type` *nazwa*

Nazwa to nazwa nowego typu. Lista deklaracji zmiennych to sekwencja deklaracji typów zmiennych będących składowymi struktury. Składowymi mogą być tablice (ale deklarowane statycznie) a także zmienne typów zdefiniowanych wcześniej.

Po zdefiniowaniu nowego typu możemy go wykorzystać w deklaracjach zmiennych postaci:

`type` (*nazwa*) *lista_nazw_zmiennych*

Po słowie `type` podajemy w nawiasie nazwę typu a później listę nazw zmiennych, które mają być tego typu.

Do poszczególnych składowych struktury możemy się odwołać łącząc znakiem % nazwę zmiennej i nazwę składowej (jakiej użyto w definicji typu). Zatem w poprzednim podrozdziale zapis `okrag2%x0` oznacza składową `x0` zmiennej `okrag2` (która to zmienna jest typu `circle`). Składowe struktury mogą być używane tak, jak zmienne odpowiedniego typu. W szczególności składowa `x0` typu `circle` jest zmienną rzeczywistą zatem skoro `okrag2` jest typu `circle` to `okrag2%x0` może pojawić się w instrukcjach tam, gdzie dozwolone jest użycie zmiennej rzeczywistej.

Przypisanie wartości wszystkim składowym struktury z wykorzystaniem instrukcji przypisania dla poszczególnych składowych byłoby uciążliwe. Możemy wykorzystać wtedy konstruktor struktury postaci:

nazwa(*lista wyrażeń*)

Nazwa to nazwa typu a lista wyrażeń zawiera wartości nadawane poszczególnym składowym w kolejności takiej, jakiej użyto w definicji typu. W razie potrzeby wyrażenia są konwertowane do odpowiedniego typu.
Jeśli zatem `x` jest zmienną rzeczywistą a `okrag3` zmienną typu `circle`, to poprawne są instrukcje

```
x = 1.0
okrag3 = circle(x,1,5.0)
```

W wyniku ich wykonania składowe `x0`, `y0` i `r` zmiennej `okrag3` przyjmą wartości odpowiednio 1.0, 1.0 (tu nastąpiła konwersja 1 całkowitej do rzeczywistej) oraz 5.0

Zmienną definiowanego typu można użyć w instrukcji przypisania do innej zmiennej tego typu. Odpowiada to instrukcji przypisania dla odpowiadających sobie składowych, np.:

```
okrag2 = okrag1
```

oznacza, że składowa `x0` zmiennej `okrag2` przyjmie wartość zapisaną w składowej `x0` zmiennej `okrag2`, itd. dla składowej `y0` i `r`.

Przy użyciu zmiennej definiowanego typu w instrukcji wejścia/wyjścia, np:

```
read(*,*) okrag1

write(*,*) okrag2
```

należy pamiętać, że składowe są czytane/wypisywane w kolejności zgodnej z kolejnością ich użycia w definicji typu (czyli tutaj: `x0`, `y0`, `r`).

9.3. I jeszcze jeden przykład

W definicji typu `circle` wszystkie składowe były tego samego typu. Oczywiście tak być nie musi. Zobaczmy to na przykładzie typu `element`:

```
type element
     integer l_at
     character(20) nazwa
     character(2) symb
     real m_at
end type element
```

Łatwo się zorientować, że zmiennych tego typu możemy użyć do zapisania informacji o pierwiastkach chemicznych:

```
type (element) wodor, wegiel, lit

wodor = element(1,'wodór','H',1.00794)
wegiel = element(6,'węgiel','C',12.0107)
lit = element(3,'lit','Li',6.941)
```

Zauważmy też, że możemy mieć tablicę o elementach zdefiniowanego typu:

```
type (element) pierwiastek(111)
```

(dalsze pierwiastki mają na razie 3-literowe symbole, więc sobie je darujmy)
i przy takich deklaracjach poprawne są instrukcje:

```
pierwiastek(1) = wodor
pierwiastek(3) = lit
```

10. Moduły

10.1. Organizacja i wykorzystanie modułów.

Jak dowiedzieliśmy się w rozdziałach poprzednich, w Fortranie 90 możemy definiować własne typy zmiennych oraz podprogramy. W praktyce często wygodne jest zgrupowanie takich definicji w jednym miejscu a następnie nadawanie dostępu do nich poszczególnym częściom programu. Służą do tego moduły.

Podejście takie ma wiele zalet. Zapewnia np. zgodność definicji w różnych częściach programu. Pozwala na tworzenie zmiennych globalnych, tzn. takich, które dostępne są wszystkim częściom programu. Jednocześnie umożliwia na ukrycie części definicji co zmniejsza ryzyko popełnienia przypadkowych błędów. Wreszcie zgrupowanie zmiennych i podprogramów z nich korzystających w jednym miejscu ułatwia tworzenie bibliotek podprogramów.

Moduł jest osobną jednostką (*unit*) programu w Fortranie 90, w związku z czym może być (choć nie musi) zapisany w innym pliku niż program główny. Zauważmy, że zapis modułu w osobnym pliku ułatwia jego wykorzystanie przez różne programy główne.

Struktura modułu jest następująca

`module` *nazwamodułu*

część specyfikacyjna

`contains`

część definiująca podprogramy

`end module` *nazwamodułu*

W części specyfikacyjnej umieszczamy deklaracje zmiennych, natomiast część następująca po słowie `contains` zawiera treść podprogramów definiowanych przez moduł. Jeśli nie definiujemy podprogramów, a moduł służy jedynie do zadeklarowania zmiennych globalnych, to pomijamy `contains` (moduł ma wtedy tylko część specyfikacyjną). Część definiująca podprogramy wygląda analogicznie, jak w przypadku podprogramów wewnętrznych omawianych w rozdziale 7. Jedyną różnicą jest to, że podprogramy zdefiniowane w module same mogą zawierać podprogramy wewnętrzne.

Innym częściom programu dajemy dostęp do zmiennych i podprogramów zdefiniowanych w module przy użyciu polecenia `use`. Ma ono składnię następującą:

`use` *nazwamodułu*

Instrukcja (lub instrukcje) `use` muszą znaleźć się na samym początku programu (lub podprogramu) wykorzystującego moduł, przed instrukcją `implicit none` oraz deklaracjami zmiennych.

W wyniku wykonania polecenia `use` program (podprogram) uzyskuje dostęp do podprogramów i zmiennych zdefiniowanych w module. Tę metodę dostępu do zmiennych nazywa się *use association* (porównaj do *host association* omawianego w rozdziale 7). Zasadnicza różnica polega na tym, że zmiennej globalnej udostępnionej lokalnie przez *use association* nie można przedefiniować lokalnie (używając deklaracji w programie używającym moduł)

Używając modułów możemy doprowadzić do problemów z konfliktem nazw. Otóż gdyby w wyniku użycia dwu modułów ta sama nazwa wskazywała na różne zmienne, to nie można się do niej odwoływać.

Pozostaje teraz powiedzieć, jak kompilować program używający moduły.
Przy kompilacji modułów kompilator generuje pomocnicze pliki o nazwach *nazwamodulu*.mod. Pliki te muszą być dostępne dla kompilatora podczas kompilacji programu używającego moduł (w chwili, gdy kompilator trafia na polecenie `use` *nazwamodulu* musi mieć już gotowy plik *nazwamodulu*.mod).

W praktyce oznacza to, że:
Jeśli treść modułów zapisana jest w tym samym pliku, co program główny, to musi poprzedzać ten program; poza tym w kompilacji nie ma różnicy w stosunku do poleceń, jakich używaliśmy dotąd.
Jeśli moduły zapisane są w osobnym pliku (plikach), to kompilator musi dostać nazwy wszystkich plików, które trzeba przetłumaczyć, np. jeśli program główny znajduje się w pliku „okaz.f90” a moduły w ”moduly.f90”, używamy polecenia:

```
gfortran –o okaz.x moduly.f90 okaz.f90
```

Zauważmy, że nazwa pliku z modułami pojawia się przed nazwą pliku z programem głównym, dzięki czemu plik ten zostanie skompilowany wcześniej.

10.2. A teraz przykład

Stwórzmy sobie moduł definiujący strukturę typu `t_element` (patrz rozdział poprzedni) oraz podprogramy operujące na zdefiniowanych zmiennych.

```
module m_molekula

  implicit none

  type t_element
    integer l_at
    character(20) nazwa
    character(2) symbol
    real m_at
  end type t_element

  type(t_element) pierwiastek(111)

  type t_atom
     integer l_at
     real coord(3)
  end type
```

```
  type(t_atom),allocatable :: molekula(:)

  integer n_at

contains

  subroutine init_uklad()

    pierwiastek(1)=t_element(1,'wodór','H',1.00794)
    pierwiastek(6)=t_element(6,'węgiel','C',12.0107)
       ! tu trzeba dodac pozostalych 109 definicji ...

  end subroutine init_uklad


  subroutine read_mol(nazwa)

    integer i,j
    character(*) nazwa

    open(11,file=nazwa)
    read(11,*) n_at
    allocate(molekula(n_at))
    do i=1,n_at
       read(11,*) molekula(i)%l_at,(molekula(i)%coord(j),j=1,3)
    end do
    close(11)
  end subroutine read_mol

  real function masa_cz()
    integer i

    masa_cz=0
    do i=1,n_at
       masa_cz=masa_cz+pierwiastek(molekula(i)%l_at)%m_at
    end do
  end function masa_cz

end module m_molekula
```

Zmienna typu `t_element` przechowuje informacje o atomie. Tablica `pierwiastek` służy do przechowania danych pierwiastków. Zwróćmy uwagę, że jej zapełnienie realizuje podprogram `init_uklad` (przykładowo tylko dla dwu pierwiastków). Zmienna typu `t_atom` przechowuje informacje o liczbie atomowej i 3 współrzędnych określających położenie tego atomu. Tablica `molekula` zawiera zatem informację o atomach (liczby atomowe i współrzędne budujących cząsteczkę). Tablica ta oraz zmienna `n_at` przechowująca liczbę atomów w cząsteczce są dostępne dla wszystkich podprogramów definiowanych w module i będą dostępne programom używającym moduł. Zwróćmy uwagę, na dodawanie przedrostka `t_` do nazw typów a `m_` do nazwy modułu. Dzięki takiej konwencji nazewnictwa nie „tracimy" przydatnych nazw (np. moduł dotyczy operacji na molekule, stąd nazwa `m_molekula`; gdybyśmy nazwali go po prostu molekula, to nie moglibyśmy użyć tej nazwy dla zmiennej)
Podprogram `read_mol` wczytuje z pliki mol.dat liczbę atomów w cząsteczce a później ich liczby atomowe i położenia. Jak nietrudno się domyślić funkcja `masa_cz` zwraca masę cząsteczki obliczoną przez zsumowanie mas atomów. Rozszyfrujmy odwołania do struktur pojawiające się w instrukcji przypisania w tej funkcji.

Zmienna `i` numeruje elementy tablicy `molekula` (będące zmiennymi typu `t_atom`). `molekula(i)` to `i`-ty element tej tablicy a `molekula(i)%l_at` to jego składowa `l_at` przechowująca liczbę atomową danego atomu. Zatem informacji o tym atomie należy szukać w odpowiednim elemencie tablicy `pierwiastek` (czyli `pierwiastek(molekula(i)%l_at)`) a interesująca nas składowa tej zmiennej przechowująca masę atomową to `pierwiastek(molekula(i)%l_at)%m_at` .

Program, który używa naszego modułu do wczytania geometrii cząsteczki i policzenia jej masy może wyglądać następująco:

```
program zwiazek
  use m_molekula
  implicit none
  character*20 plik

  call init_uklad()
  write(*,*) 'podaj nazwe pliku z danymi'
  read(*,'(a)') plik
  call read_mol(plik)
  write(*,*) 'masa czasteczkowa = ',masa_cz()

end program zwiazek
```

11. Przykładowe programy z wykorzystaniem podprogramów bibliotecznych

W rozdziale tym zobaczymy, jak konstruować własne programy wykorzystując podprogramy biblioteczne.

11.1. Wykorzystanie funkcji generującej liczby pseudolosowe

W wielu przypadkach, szczególnie przy programowaniu symulacji, musimy wygenerować w programie ciąg liczb losowych. Teoretycznie program działający przecież w sposób ściśle deterministyczny ciągu takiego wyprodukować nie może, chyba że kontaktuje się z otoczeniem będącym źródłem losowego szumu. W systemach operacyjnych Unix/Linux informacje o przypadkowych sygnałach płynących z otoczenia są zbierane (źródło entropii) i mogą posłużyć do generowania liczb losowych np. dla celów kryptograficznych.

W praktyce dla celów symulowania procesów losowych używane są ciągi liczb pseudolosowych – jakkolwiek generowane są deterministycznie (przy pomocy funkcji dających dostatecznie „chaotyczne" wartości) to spełniają wymagane przez nas testy dotyczące losowości. Tym niemniej przy użyciu bardziej wyrafinowanych testów moglibyśmy stwierdzić odchylenia od pełnej przypadkowości. Pamiętać także należy, że generatory liczb pseudolosowych mają okresy powtarzalności – po wyprodukowaniu dostatecznie wielu liczb ich ciąg zaczyna się powtarzać. Nietrudno się domyślić, iż generator liczb pseudolosowych jest tym lepszy, im bardziej wyrafinowane testy losowości spełnia produkowany przez niego ciąg liczb i im dłuższy jest okres powtórzeń.
Ciąg liczb pseudolosowych tworzonych przez generator zależy od pewnych parametrów startowych tego generatora (*seed*). Generator uruchomiony z tą samą wartością początkową wygeneruje ten sam ciąg liczb, co umożliwia uzyskiwanie powtarzalnych wyników. Aby różne wykonania programu dawały różne ciągli liczb pseudolosowych, musimy zmieniać wartość *seed* (np. wykorzystując zegar systemowy, lub źródło entropii).

W Fortranie 90 dostępna jest procedura `random_number` zwracająca liczbę rzeczywistą z przedziału otwartego od 0 do 1. Liczby te generowane są według równomiernego rozkładu prawdopodobieństwa (tzn. każda wartość z tego przedziału jest jednakowo prawdopodobna).

Przed użyciem procedury `random_number` wskazana jest inicjalizacja generatora liczb losowych. Służy do tego procedura `random_seed`. Sposób wyspecyfikowania wartości inicjujących generator w przypadku tej procedury jest dość skomplikowany; na tym etapie wystarczy nam informacja, iż wywołanie

```
call random_seed()
```

powinno zainicjować generator wartością zależną od konkretnego systemu i kompilatora.

Jak się niestety okazuje, w przypadku kompilatora gfortran będzie to zawsze ta sama wartość. Aby temu zaradzić, skorzystajmy z napisanej *ad hoc* procedury `init_random_seed` zawartej w module `initrs` (plik initrs.f90) na stronie z materiałami do ćwiczeń.

Argumentem procedury `random_number` jest zmienna lub tablica typu `real`. W wyniku wywołania procedury zmiennej tej lub wszystkim elementom tablicy nadane zostają losowe wartości.

Napiszmy program, który wygeneruje ciąg liczb o podanej długości. Liczby powinny być losowane równomiernie z przedziału (a,b) – zatem musimy przeskalować wartości zwracane przez procedurę `random_number`.

Przykładowy program może wyglądać następująco:

```
program rantest
use initrs
implicit none
integer n,i
real a,b,x,w

write(*,*) 'podaj a i b'
read(*,*) a,b
write(*,*) 'ile liczb losowac? '
read(*,*) n

w=b-a

open(11,file='random.dat')

call init_random_seed()

do i=1,n
  call random_number(x)
  x=a+w*x
  write(11,*) x
end do

close(11)
end program rantest
```

Zauważmy, że wobec faktu, iż procedura może zwrócić tablicę, program mógłby wyglądać następująco:

```
program rantest
use initrs
implicit none
integer n
real a,b,w
real, allocatable :: x(:)

write(*,*) 'podaj a i b'
read(*,*) a,b
write(*,*) 'ile liczb losowac? '
read(*,*) n

w=b-a

open(11,file='random.dat')
```

```
call init_random_seed()

allocate(x(n))
call random_number(x)
x=a+w*x
do i=1,n
     write(11,*) x(i)
end do
deallocate(x)

close(11)
end program rantest
```

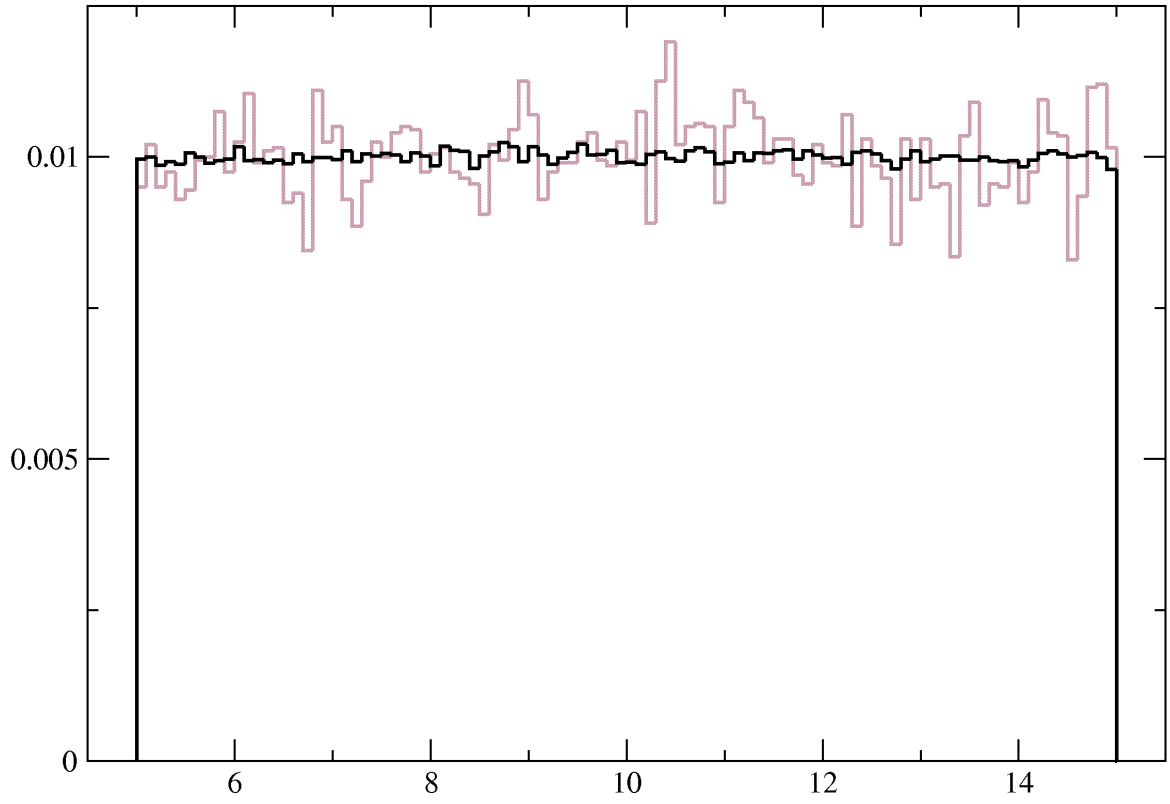
W instrukcji `x=a+w*x` wykorzystaliśmy zasady działań na macierzach w Fortranie.
Zwróćmy jednak uwagę, że wcale nie trzeba pamiętać wszystkich generowanych liczb (wystarczy wypisywać je pojedynczo), zatem nasz zmodyfikowany program zupełnie niepotrzebnie zużywa pamięć komputera na alokowanie tablicy `x`.

Zakładając, że program zapisany jest w pliku rantesf.f90 skompilujemy go (wraz ze ściągniętym ze strony plikiem initrs.f90) poleceniem:

```
gfortran –o rantest.x initrs.f90 rantest.f90
```

Zadanie. Przetestuj działanie programu generując serie liczb o różnej długości i tworząc przy pomocy jakiegoś programu do obróbki danych (np. Xmgrace) histogramy ich rozkładu.

Wyniki powinny być zbliżone do przykładu na poniższym rysunku, gdzie przedstawiono histogramy dla serii 20000 (krzywa brązowa) i 1000000 (krzywa czarna) liczb losowanych w przedziale (5; 15). Przy tworzeniu histogramu podzielono ten przedział na 100 części, zatem w idealnym przypadku w każdym przedziale powinien znaleźć się 1 % wszystkich liczb. Widać, że przy zwiększaniu próbki zbliżamy się do tego wyniku.



11.2. Rozwiązywanie układu równań liniowych (teoria)

Układ N równań liniowych z N niewiadomymi możemy zapisać następująco:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1N}x_N &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2N}x_N &= b_2 \\ &\vdots \\ a_{N1}x_1 + a_{N2}x_2 + \ldots + a_{NN}x_N &= b_N \end{aligned},$$

czyli w postaci macierzowej

$$\mathbf{A}\cdot\mathbf{x} = \mathbf{b},$$

gdzie $\mathbf{A} = \begin{bmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{N1} & \cdots & a_{NN} \end{bmatrix}$ to macierz współczynników w równaniach, $\mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$ to wektor kolumnowy prawych stron równań (znany) a $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix}$ to kolumnowy wektor zmiennych występujących w równaniach (poszukiwany).

Nietrudno zauważyć, że znając macierz odwrotną do **A**, czyli $\mathbf{A^{-1}}$ łatwo (teoretycznie) otrzymujemy szukane rozwiązanie:

$$\mathbf{x} = \mathbf{A^{-1}} \cdot \mathbf{b}$$

W praktyce znalezienie rozwiązania w ten sposób jest niewykonalne numerycznie, wymaga bowiem obliczania wyznaczników. Tymczasem ilość mnożeń potrzebna do obliczenia wyznacznika macierzy $N \times N$ jest rzędu $N!$, czyli bardzo szybko rośnie wraz ze wzrostem wymiaru rozwiązywanego układu.

Rozwiązanie układu równań znajduje się wykorzystując metodę eliminacji Gaussa. Zauważmy, że dodając stronami do równań 1 do N-1 równanie ostatnie pomnożone przez odpowiednie współczynniki otrzymamy równania, w których nie występuje zmienna x_N. Postępując analogicznie możemy wyeliminować zmienną x_{N-1} z równań 1 do N-2, itd. Ostatecznie przekształcamy macierz **A** do postaci:

$$\begin{bmatrix} \bullet & 0 & 0 & 0 \\ \bullet & \bullet & 0 & 0 \\ \bullet & \bullet & \bullet & 0 \\ \bullet & \bullet & \bullet & \bullet \end{bmatrix},$$

w której wszystkie niezerowe elementy znajdują się na i pod główną przekątną. Mówimy, że macierz **A** sprowadziliśmy do postaci trójkątnej dolnej. Nietrudno teraz z pierwszego równania obliczyć x_1, podstawiając obliczone x_1 do równania drugiego obliczyć x_2, itd.

Zauważmy, że operacje, jakie musieliśmy wykonać, aby sprowadzić **A** do postaci trójkątnej dolnej nie zależą od wektora prawych stron równań **b**. Oznacza to, że niewiele większym wysiłkiem będziemy mogli rozwiązać nie jeden a wiele układów równań opisanych tą samą macierzą **A**, a różniących się tylko prawymi stronami.

Dla porządku powiedzmy jeszcze, jak rozwiązywanie układu równań liniowych wykorzystać do obliczenia wyznacznika lub macierzy odwrotnej.

Przypuśćmy, że dla zadanej macierzy **A** rozwiązaliśmy N układów równań o prawych stronach

$$\mathbf{b^{(1)}} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \mathbf{b^{(2)}} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \quad \ldots \quad , \quad \mathbf{b^{(N)}} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

a otrzymane wektory rozwiązań $\mathbf{x^{(1)}}$, $\mathbf{x^{(2)}}$, ... ,$\mathbf{x^{(N)}}$ ustawiliśmy obok siebie, jako kolumny macierzy $N \times N$. Nietrudno zauważyć, że otrzymana macierz to $\mathbf{A^{-1}}$.

Operacje, jakie wykonywaliśmy sprowadzając macierz do postaci trójkątnej dolnej (mnożenie wierszy przez liczbę, dodawanie wierszy stronami) nie zmieniają jej wyznacznika. Jest on zatem taki sam, jak wyznacznik otrzymanej macierzy trójkątnej dolnej. Ten zaś łatwo obliczyć jako iloczyn elementów na głównej przekątnej. W praktyce eliminację zmiennych dokonuje się niekoniecznie w kolejności, w jakiej zostały ponumerowane. Zmiana kolejności

odpowiada zamianie dwu wierszy macierzy (czyli zmienia znak jej wyznacznika). Oznacza to, że w praktyce musimy jeszcze wiedzieć, czy ilość takich zmian była parzysta, czy nieparzysta, aby pomnożyć iloczyn elementów na przekątnej przez 1 lub –1.

11.3. Rozwiązywanie układu równań liniowych (program)

Przy rozwiązywaniu układu równań liniowych wykorzystamy powszechnie stosowane w algebrze liniowej biblioteki LAPACK (Linear Algebra PACKage) i Lapack95 (interfejs Lapacka do Fortranu 95) oraz BLAS (Basic Linear Algebra Solutions). Biblioteki te w wersji binarnej są dostępne z http://www.netlib.org/, stamtąd też można uzyskać kody źródłowe podprogramów.

Potrzebny nam podprogram to procedura `la_gesv`. Na podstawie podręcznika ustalamy argumenty podprogramu:

```
SUBROUTINE LA_GESV(A,B,IPIV,INFO)
```

`a` – dwuwymiarowa tablica podwójnej dokładności przechowująca współczynniki równań. Tablica ta ma kształt `n×n`, gdzie `n` to liczba równań.
`b` - dwuwymiarowa tablica o wymiarach `n×nrhs` przechowująca w kolejnych kolumnach współczynniki prawych stron kolejnych układów równań (stron tych jest `nrhs`)
`ipiv` – jednowymiarowa tablica całkowita mająca `n` elementów
`info` – zmienna całkowita

Przed wywołaniem podprogramu musimy zapisać w tablicy `a` współczynniki układu równań a w tablicy `b` wartości prawych stron rozwiązywanych układów.
Po zakończeniu pracy podprogramu, zmienna `info` przyjmuje wartość 0, jeśli nie wystąpiły problemy, lub wartość różną od 0 w przeciwnym wypadku. Jeśli praca procedury przebiegła pomyślnie to tablica `a` zawiera postać trójkątną dolną wyjściowej macierzy, kolejne kolumny macierzy zapisanej w tablicy `b` to rozwiązania układu dla kolejnych prawych stron a tablica `ipiv` zawiera informacje o przestawianiu wierszy macierzy przy eliminacji. Nam wystarczą dane zwrócone w tablicy `b`.

Zobaczmy wobec tego, jak może wyglądać kod programu korzystającego z procedury `la_gesv`:

```
program urlin
  use la_precision
  use f95_lapack
  implicit none
  integer info
  real(dp), allocatable :: a(:,:),b(:,:)
  integer, allocatable :: ipiv(:)

  call czytaj()

  call la_gesv(a,b,ipiv,info)
```

```
  if (info==0) then
     call wypisz()
  else
     write(*,*) 'problemy'
  end if

  deallocate(a,b,ipiv)

contains

  subroutine czytaj()
    integer i,j,n,m

    open(11,file='rown.dat')
    read(11,*) n,m
    allocate(a(n,n),b(n,m),ipiv(n))
    do i=1,n
       read(11,*) (a(i,j),j=1,n)
    end do
    do i=1,n
       read(11,*) (b(i,j),j=1,m)
    end do
    close(11)
  end subroutine czytaj

  subroutine wypisz()
    integer i,j

    open(11,file='rown.res')
    do i=1,size(b,1)
       write(11,*) (b(i,j),j=1,size(b,2))
    end do

  end subroutine wypisz

end program urlin
```

Polecenia `use` udostępniają definicje zawarte w modułach. W module `la_precision` zdefiniowana jest zmienna `dp`, która przechowuje informacje o rodzaju zmiennej rzeczywistej podwójnej dokładności. Używamy jej w deklaracji tablic określając typ ich elementów jako `real(dp)`.
Moduł `f95_lapack` udostępnia procedurę `la_gesv`.
Tablice `a`, `b` i `ipiv` deklarujemy jako alokowalne, ich alokacji dokonujemy po wczytaniu informacji o rozmiarze rozwiązywanego układu.
Dla przejrzystości czytanie danych i wypisywanie wyników delegowaliśmy do podprogramów `czytaj` i `wypisz`.
Zmienne `n` i `m` przechowują aktualny wymiar naszego układu i ilość prawych stron. Zmienne te i tablice `a` i `b` są czytane przez podprogram `czytaj`.

Rozwiązania naszych układów równań zawarte są w tablicy `b`, dlatego tylko ją wypisujemy. Wywołanie funkcji `size(b,1)` lub `size(b,2)` podaje odpowiednio liczbę wartości pierwszego lub drugiego indeksu tablicy `b`.

Zadanie. Na podstawie kodu podprogramu `czytaj` ustal, jak w pliku 'rown.dat' mają być zapisane dane. Jak należy czytać plik wynikowy?

Pozostaje jeszcze skompilować nasz program.
Aby kompilacja się powiodła, musimy mieć zainstalowane biblioteki BLAS, Lapack i Lapack95. Przy kompilacji musimy zażądać od kompilatora dołączenia do naszego programu podprogramów z tych bibliotek (potrzebujemy BLAS-a, ponieważ jest wykorzystywany przez Lapack-a). Służy do tego parametr `-l`, po którym (bez spacji) podajemy nazwę żądanej biblioteki: `-l`*nazwabib*. Dodatkowo musimy powiedzieć kompilatorowi, gdzie ma szukać plików .mod zawierających informacje o modułach (parametr `-I`, po którym podajemy lokalizację odpowiedniej kartoteki).
W naszym przypadku (konfiguracja w pracowni studenckiej) skompilujemy program poleceniem:

```
gfortran -o urlin.x urlin.f -I/usr/lib/mod -llapack95 -llapack
-lblas
```

(`-lblas` moglibyśmy pominąć, bo jest dołączany domyślnie)

Zadanie. Przerób program tak, aby wczytywał tylko macierz **A** o wymiarze n×n i obliczał jej macierz odwrotną. Zauważ, że wymaga to skonstruowania jednostkowej macierzy **B** (jedynki na przekątnej, reszta elementów to zera) i przekazania jej jako macierz n prawych stron układu. Po zadziałaniu procedury `la_gesv` macierz **B** będzie zawierać macierz odwrotną do **A** (patrz 11.2)

Uzupełnienie A. Wybrane podprogramy wewnętrzne Fortranu.

W dodatku A wymienimy część z podprogramów zdefiniowanych w standardzie Fortranu. Są to podprogramy „wewnętrzne" – dostępne są w każdej części programu i nie wymagają deklaracji (deklaracja oznaczałaby chęć zastąpienia podprogramu standardowego innym, definiowanym przez programistę).

A1. Funkcje

Funkcje obliczające wartości funkcji matematycznych:

`abs` - wartość bezwzględna argumentu
`exp` - funkcja wykładnicza
`log` - logarytm naturalny
`log10` - logarytm dziesiętny
`sqrt` - pierwiastek kwadratowy
`cos` - cosinus
`sin` - sinus
`tan` - tangens
`acos` - arcus cosinus
`asin` - arcus sinus
`atan` - arcus tangens

Argument wymienionych wyżej funkcji (oprócz `abs`) nie może być całkowity. We wszystkich przypadkach argumentem może być skalar lub tablica. W tym drugim przypadku wynikiem działania funkcji jest tablica o zgodnych wymiarach, której elementami są wyniki działania funkcji na elementy tablicy wyjściowej. Na przykład, jeśli `a` i `b` są jednowymiarowymi tablicami o trzech elementach, to po wykonaniu instrukcji

```
a(1) = 4.0
a(2) = 9.0
a(3) = 16.0
b = sqrt(a)
```

kolejne elementy tablicy `b` przyjmą wartości 2.0, 3.0 i 4.0.

Funkcje znajdujące ekstrema:

`min` - wartość najmniejsza
`max` - wartość największa

Funkcje te znajdują ekstremalną wartość wśród swoich argumentów. Argumenty muszą być tego samego typu, jest to równocześnie typ wyniku funkcji, np.

`min(1,-3,5,8)` ma wartość –3
`max(3.7,-1.9,8.4)` ma wartość 8.4

Także w przypadku tych funkcji argumentami mogą być tablice o zgodnych rozmiarach.

Reszta z dzielenia:
`mod(`*arg1*`,`*arg2*`)` - reszta z dzielenia *arg1* przez *arg2*
Funkcje konwersji typów:

`int` - konwersja (przez obcięcie) na wartość całkowitą (wynik jest całkowity), np. `int(4.6)` wynosi 4
`aint` - jak wyżej, ale wynik jest rzeczywisty, np. `aint(4.6)` wynosi 4.0
`nint` - zaokrąglenie wartości rzeczywistej do całkowitej (wynik całkowity), np. `nint(4.6)` wynosi 5
`anint` - jak wyżej, ale wynik jest rzeczywisty, np. `anint(4.6)` wynosi 5.0
`real` - konwersja na wartość rzeczywistą, np. `real(1)` wynosi 1.0
`cmplx` - konwersja na wartość zespoloną, np. `cmplx(2)` lub `cmplx(2.0)` wynosi (2.0, 0.0)
`conjg` - liczba sprzężona do argumentu (zespolonego), np. `conjg(1.0,1.0)` wynosi (1.0,-1.0)

Funkcja zwracająca rozmiar tablicy:

`size(`*tablica*`, `*wymiar*`)` - wywołana jako `size(`*tablica*`)` zwraca liczbę elementów tablicy; wywołana jako `size(`*tablica*`, `*wymiar*`)` zwraca liczbę wartości przyjmowanych przez indeks w danym wymiarze; np. jeśli `a` jest tablicą 3×5, to `size(a,1)` wynosi 3, `size(a,2)` jest równe 5 a `size(a)` wynosi 15

Funkcje operujące na macierzach i wektorach:

`dot_product` - jej argumentem są dwie jednowymiarowe tablice o zgodnym rozmiarze, wynikiem iloczyn skalarny tych dwu wektorów

`matmul` - jeśli pierwszy argument to tablica o wymiarach n×k a drugi tablica o wymiarach k×m to wynikiem jest tablica n×k będąca iloczynem macierzowym argumentów

`transpose` - argumentem jest dwuwymiarowa tablica, wynikiem tablica transponowana

A2. Procedury

`random_seed` - inicjuje generator liczb pseudolosowych; zawołana bez argumentu:
`call random_seed()`
powinna inicjować generator przypadkowo wybraną wartością

`random_number` - argumentem jest zmienna lub tablica typu rzeczywistego, w wyniku wykonania procedury zmienna ta lub elementy tablicy przyjmują wartości liczb pseudolosowych o rozkładzie prostokątnym z przedziału <0,1)