

Spis treści

1	Erlang	2
1.1	Wartościowanie - rodzaje	2
1.2	Typy danych: atom, liczba, t. funkcyjny, pid, krotka, lista, rekord, term	2
1.3	Generatory list (List Comprehensions)	3
1.4	Funkcje z modułu lists	3
1.5	Wyjątki - rodzaje i składnia	4
1.6	Funkcja, w tym fun. z dozorem (warunek), rekurencja	5
1.7	Instrukcje warunkowe if i case	5
1.8	BONUS - procesy i komunikacja między procesami	6
2	Ada	7
2.1	Klasyfikacja typów	7
2.2	Definiowanie typów i podtypów	8
2.2.1	Definiowanie typów	8
2.2.2	Podtypy	8
2.3	Podprogramy - procedury i funkcje	9
2.4	Rekordy	12
2.5	Tablice	13
2.6	Kwantyfikatory	15
2.7	Instrukcje sterujące	15
2.8	Definicje i zastosowania pojęć jak: atrybuty, agregaty, aspekty, pragmy, wyróżniki	17
2.8.1	Atrybuty	17
2.8.2	Agregaty	18
2.8.3	Pragmy, Aspekty	18
2.9	Wskaźniki i ich rodzaje. Element listy. Wskaźniki do podprogramów.	18
2.9.1	new keyword	19
2.9.2	Dereferencja wskaźników	19
2.10	Pakiety	20
2.11	Wyjątki	22
2.12	Zadania i typy zadaniowe. Tworzenie i kończenie zadań.	23
2.13	Zmienne dzielone	24
2.14	Obiekt chroniony	25
3	Wiedza ogólna	26
3.1	Proces i jego stany. Szeregowanie. Przeplot. Priorytety.	26
3.2	Współbieżność a równoległość.	27
3.3	Wzajemne wykluczanie. Sekcja krytyczna.	27
3.4	Metody synchronizacji i komunikacji procesów, w tym semafor, komunikaty, monitory, spotkania.	27
3.5	Spotkania i ich rodzaje.	28
3.6	Komunikacja synchroniczna i asynchroniczna.	28
3.7	Proces/wątek - podobieństwa i różnice.	28

1 Erlang

1.1. Wartościowanie - rodzaje

W Erlangu można korzystać z dwóch rodzajów wartościowania:

- zachłanne (eager evaluation) - kiedy argumenty funkcji są wyznaczane przed jej wywołaniem, zaletą tego podejścia jest to, że można określić kolejność wykonywania obliczeń, a wadą narzut związany z wykonywaniem tych obliczeń
- leniwe (lazy evaluation) - kiedy argumenty funkcji wyznaczane są tylko wtedy kiedy są potrzebne, na żądanie, takie podejście daje również możliwość tworzenia nieskończonych struktur danych

Pytanie

Jakie rodzaje wartościowania są w Erlangu ?

1.2. Typy danych: atom, liczba, t. funkcyjny, pid, krotka, lista, rekord, term

Erlang jest językiem dynamicznie typowanym, typy są przypisywane zmiennym w trakcie działania programu na podstawie ich wartości, nie są znane na etapie kompilacji. W Erlangu wyróżniamy następujące typy danych:

- Term - dana o jakimkolwiek typie nazywana jest termem
- Number - integer lub float, liczbę możemy zapisywać normalnie lub jako *base#value*, gdzie base to podstawa systemu liczbowego

```
1 1> 42.  
2 42  
3 2> $A.  
4 65  
5 4> 2#101.  
6 5
```

- Atom - stała z nazwą, zaczyna się z małej litery bez znaków specjalnych, a jeśli nie to musi być ujęta w apostrofy, np. *sample*, *'Luty'*.

```
1 hello  
2 phone_number  
3 'Monday'  
4 'phone number'
```

- Fun - to zmienna do której przypisujemy funkcję, dzięki temu możemy ją np. przekazywać do innej funkcji

```
1 1> Fun1 = fun (X) -> X+1 end  
2 #Fun<erl_eval.6.39074546>
```

- Pid - to typ, który przechowuje pid procesu składający się z trzech liczb

```
1 1> spawn(m, f, []).  
2 <0.51.0>
```

- Krotka - krotka to złożony typ danych składający się z niezmiennej liczby termów

```
1 1> P = {adam,24,{july,29}}.  
2 {adam,24,{july,29}}
```

- Lista - to złożony typ danych, który przechowuje zmienną liczbę termów, pierwszy element listy to głowa, z reszta listy to tzw. ogon

```
1 1> L1 = [a,2,{c,4}].  
2 [a,2,{c,4}]  
3 2> [H|T] = L1.  
4 [a,2,{c,4}]
```

- Rekord - to złożony typ danych, który przechowuje niezmienną liczbą termów, rekord ma nazywane pola, jest podobny do struktury z języka C

```
1 -record(person, {name, age}).  
2  
3 new(Name, Age) ->  
4   #person{name=Name, age=Age}.
```

- Inne - oprócz tego w Erlangu mamy też typy danych: Boolean, Reference, Map... więcej na stronie http://erlang.org/doc/reference_manual/data_types.html#reference
- UWAGA: W Erlangu nie mamy typu String jako takiego, Stringi to tak naprawdę listy zawierające symbole jako elementy listy.
- WAŻNE: W Erlangu zmienne zapisujemy dużymi literami, a stałe małymi literami

Pytanie

Jakie typy danych są w Erlangu ? Jakim rodzajem języka programowania jest Erlang względem typów zmiennych ?

1.3. Generatory list (List Comprehensions)

List comprehensions służą do generacji list, składają się z przypisania i predykatów, pozwalają na wygenerowanie listy wartości, które spełniają określone warunki.

```
1 > [X || X <- [1,2,a,3,4,b,5,6], X > 3].  
2 [a,4,b,5,6]  
3 > [X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].  
4 [4,5,6]
```

Pytanie

Czym są Generatory List (List Comprehensions) w Erlangu ?

1.4. Funkcje z modułu lists

Moduł lists zawiera wiele użytecznych funkcji, które działają na listach, jest ich bardzo dużo, dlatego po opis wszystkich należy sięgnąć do dokumentacji, przedstawmy sobie kilka użytecznych:

- *zipwith* - z dwóch list tworzy listę krotek o pierwszym elemencie z pierwszej, a drugim z drugiej listy

```
1 lists:zipwith(fun(X, Y) -> {X, Y} end, [1,2,3],[a,b,c]).
2 [{1,a},{2,b},{3,c}]
```

- *foldl* - składa listę do jednej wartości przez wykonanie określonej przez użytkownika funkcji, podajemy funkcję, wartość początkową zmiennej do której składamy listę i na końcu samą listę do poskładania

```
1 lists:foldl(fun(X, Sum) -> X + Sum end, 0, [1,2,3,4,5]).
2 15
```

- *append* - dokleja listę na koniec innej listy

```
1 lists:append([1,2,3],[a,b,c]).
2 [1,2,3,a,b,c]
```

- *map* - zmienia każdy element listy według zadanej funkcji, podajemy funkcję, a po niej listę

```
1 lists:map(fun(X)-> {X} end, [1,2,3,4]).
2 [{1},{2},{3},{4}]
```

- *mapfoldl* - połączenie map i foldl

```
1 lists:mapfoldl(fun(X, Sum) -> {2*X, X+Sum} end,
2 0, [1,2,3,4,5]).
3 {[2,4,6,8,10],15}
```

Pytanie

Omów przykładowe funkcje z modułu *lists*.

1.5. Wyjątki - rodzaje i składnia

W Erlangu mamy trzy rodzaje wyjątków:

- *error* - to wyjątki powstające wskutek wykonywania programu lub poprzez wywołanie

```
1 erlang:error(Why) % Give reason of the exception
```

Wyjątki te kończą działanie procesu

- *exit* - to wyjątki wyrzucane przez wywołanie

```
1 exit(Why) % Give reason of the exception
```

Wywołanie *exit* kończy działanie procesu i wysyła wiadomość $\{EXIT', Pid, Why\}$ do powiązanych procesów.

- *throw* - to typ wyjątków, który programista może obsłużyć w kodzie, wyjątki te nie kończą procesu, ale zmieniają przepływ wykonywania instrukcji

```
1 throw(Why) % Give reason of the exception
```

Przykładowo

```

1 im_impressed() ->
2 try
3     talk(),
4     _Knight = "None shall Pass!",
5     _Doubles = [N*2 || N <- lists:seq(1,100)],
6     throw(up),
7     _WillReturnThis = tequila
8 catch
9     Exception:Reason -> {caught, Exception, Reason}
10 end.

```

Pytanie

Omów wyjątki w Erlangu.

1.6. Funkcja, w tym fun. z dozorem (warunek), rekurencja

Funkcja w Erlangu składa się z głowy i z ciała, które są oddzielone znakiem ->.

Głowa składa się z nazwy funkcji, listy argumentów oraz opcjonalnie strażnika. Można zdefiniować wiele 'wersji' funkcji oddzielając je średnikiem, ostatnią z wersji kończymy kropką.

Funkcje w Erlangu często wywołują same siebie (rekurencja), co zastępuje nam iterowanie po liście/zakresie.

Przykład obliczania silni w Erlangu z rekurencją i strażnikami:

Listing 1: Silnia

```

1 fact(N) when N>0 -> % first clause head
2   N * fact(N-1); % first clause body
3
4 fact(0) -> % second clause head
5   1. % second clause body

```

Wywołanie funkcji w Erlangu poszukuje odpowiedniej wersji funkcji tj. takiej w której:

- Argumenty pasują do wzorca
- Spełniony jest strażnik

Pytanie

Omów funkcje w Erlangu, jak zaimplementować rekurencję ? Jak zaimplementować dozór (warunek, inaczej strażnik) ?

1.7. Instrukcje warunkowe if i case

Instrukcja *if* w Erlangu działa poprzez zastosowanie strażników. Definiujemy kilka strażników, a *if* skanuje je po kolei w poszukiwaniu pierwszego, który jest spełniony. Jeśli żaden nie jest spełniony to wyrzucany jest błąd, dlatego dobrze zawsze zapewnić strażnika *true*, który łapie wszystkie inne przypadki (działa jak *else*).

Listing 2: Przykład if

```

1 is_greater_than(X, Y) ->
2   if
3     X>Y ->
4       true;
5     true -> % works as an 'else' branch
6       false
7   end

```

Instrukcja *case* działa bardzo podobnie do *if*, ale ta wykorzystuje *Pattern Matching* zamiast strażników. Najwięcej pokaże nam przykład

Listing 3: Przykład case

```

1 is_valid_signal(Signal) ->
2   case Signal of
3     {signal, _What, _From, _To} ->
4       true;
5     {signal, _What, _To} ->
6       true;
7     _Else ->
8       false
9   end.

```

Pytanie

Omów instrukcje *if* i *case* w Erlangu.

1.8. BONUS - procesy i komunikacja między procesami

Procesy w Erlangu tworzymy przy pomocy funkcji *spawn*, która zwraca id procesu:

```

1 PID = spawn(m, f, [a])
2 % m - module name
3 % f - function name
4 % [a] - arguments list

```

Wiadomości w Erlangu wysyłamy przy użyciu symbolu wykrzyknika. Podajemy PID procesu do którego chcemy wysłać wiadomość:

```

1 PID ! {self(), message}

```

Wiadomości w procesie odbieramy poprzez użycie *receive* oraz *end*, stosujemy pattern matching aby odebrać wiadomość:

```

1 receive
2   {reset, Board} -> reset(Board);
3   _Other -> {error, unknown_msg}
4 end


```

Pytanie

Omów tworzenie i przekazywanie wiadomości pomiędzy procesami w Erlang.

2 Ada

2.1. Klasyfikacja typów



Klasyfikacja typów

Typy Ady:

- I. Proste**
 - 1) skalarne**
 - (a) dyskretne**
 - wyliczeniowe
 - całkowite: ze znakiem i resztowe
 - (b) rzeczywiste**
 - zmiennopozycyjne
 - stałopozycyjne: zwykłe i dziesiętne
 - 2) wskaźnikowe**
- II. Złożone**
 - 1) tablice**
 - 2) rekordy**
 - 3) zadania**
 - 4) typy chronione**

Pytanie

Omów klasyfikację typów w Adzie

Trochę więcej informacji o typach:

- Ada jest językiem o statycznym typowaniu (w przeciwieństwie do Erlanga, który jest typowany dynamicznie), oznacza to, że typy zmiennych muszą być znane na etapie kompilacji.
- W Adzie typy zmiennych zaczynają się dużą literą
- Typy zapisujemy po dwukropku po nazwie zmiennej (patrz przykład poniżej)

Najważniejsze podstawowe typy danych w Adzie to:

- Integer - literał zapisujemy jako ciąg cyfr
- Float - literał zapisujemy jako ciąg cyfr z częścią ułamkową po kropce (kropka jest konieczna !)
- Boolean - przyjmuje wartość *True* lub *False*
- Character - stałe zapisujemy w pojedynczych apostrofach, np. 'A'
- String - literały zapisujemy w cudzysłowach
- UWAGA: Wartość do zmiennej przypisujemy za pomocą operatora `:=`

Listing 4: Silne typowanie

```

1 procedure Strong_Typing is
2   Alpha: Integer := 1;
3   Beta: Integer := 10;
4   Result: Float;
5 begin
6   Result := Float (Alpha) / Float (Beta);
7 end Strong_Typing;
```

2.2. Definiowanie typów i podtypów

2.2.1. Definiowanie typów

W Adzie możemy w prosty sposób definiować własne typy na podstawie typów wbudowanych. Tak utworzone typy możemy później używać w programie tak jak wszystkie inne. Do definiowania typu używamy słowa kluczowego *type*. Tak zdefiniowane typy nazywamy *derived types*, ponieważ w pewnym sensie dziedniczą one po istniejących w Adzie typach. Przykład:

```

1 procedure Main is
2   -- ID card number type, incompatible with Integer.
3   type Social_Security_Number
4     is new Integer range 0 .. 999_99_9999;
5   --           ^ Since a SSN has 9 digits max, and cannot be
6   --           negative, we enforce a validity constraint.
7
8   SSN : Social_Security_Number := 555_55_5555;
9   --           ^ You can put underscores as formatting in
10  --           any number.
11
12  I : Integer;
13
14  Invalid : Social_Security_Number := -1;
15  --           ^ This will cause a runtime error
16  --           (and a compile time warning with
17  --           GNAT)
18 begin
19  I := SSN;           -- Illegal, they have different types
20  SSN := I;           -- Likewise illegal
21  I := Integer (SSN); -- OK with explicit conversion
22  SSN := Social_Security_Number (I); -- Likewise OK
23 end Main;
```

Pytanie

Jak w Adzie definiujemy typy ?

2.2.2. Podtypy

Podtypy używane są w Adzie zazwyczaj w celu zawężenia ograniczeń wartości dla danego typu. Definiujemy je używając słowa kluczowego *subtype*, a następnie używamy ich w kodzie tak jak innych typów.

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet is
4   type Days is (Monday, Tuesday, Wednesday, Thursday,
5                 Friday, Saturday, Sunday);
6
```



```

7  — Declaration of a subtype
8  subtype Weekend_Days is Days range Saturday .. Sunday;
9  — ^ Constraint of the subtype
10
11  M : Days := Sunday;
12
13  S : Weekend_Days := M;
14  — No error here, Days and Weekend_Days are of the same type.
15  begin
16    for I in Days loop
17      case I is
18        — Just like a type, a subtype can be used as a
19        — range
20        when Weekend_Days =>
21          Put_Line ("Week end!");
22        when others =>
23          Put_Line ("Hello on " & Days'Image (I));
24        end case;
25      end loop;
26  end Greet;

```

2.3. Podprogramy - procedury i funkcje

W Adzie mamy dwa rodzaje podprogramów: procedury i funkcje.

Pytanie

Czym w Adzie różni się procedura od funkcji ?

Procedura nic nie zwraca, a funkcja coś zwraca.

Pierwszy przykładowy program w języku Ada z laboratorium:

Listing 5: Lab1.adb

```

1  — Lab1.adb
2  — komentarz do końca linii
3
4  — wykorzystany pakiet
5  with Ada.Text_IO;
6  use Ada.Text_IO;
7
8  — procedura główna — dowolna nazwa (ale taka jak nazwa pliku)
9  procedure Lab1 is
10
11  — część deklaracyjna
12
13  — funkcja — forma pełna
14  function Max2(A1, A2 : in Float) return Float is
15  begin
16    if A1 > A2 then return A1;
17    else return A2;
18    end if;
19  end Max2;
20
21  — funkcja wyrażeniowa
22  — forma uproszczona funkcji
23  — jej treść jest tylko wyrażeniem w nawiasie
24
25  function Add(A1, A2 : Float) return Float is
26    (A1 + A2);
27

```

```

28 function Max(A1, A2 : in Float ) return Float is
29   (if A1 > A2 then A1 else A2);
30
31 — Fibonacci
32 function Fibo(N : Natural) return Natural is
33   (if N = 0 then 1 elsif N in 1|2 then 1 else Fibo(N-1) + Fibo(N-2) );
34
35   — procedura
36   — zparametryzowany ciąg instrukcji
37 procedure Print_Fibo(N: Integer) is
38 begin
39   if N <1 or N>46 then raise Constraint_Error;
40   end if;
41   Put_Line("Ciąg Fibonacciego dla N= " & N'Img);
42   for I in 1..N loop
43     Put( Fibo(I)'Img & " " );
44   end loop;
45   New_Line;
46 end Print_Fibo;
47
48 — poniżej treść procedury głównej
49 begin
50   Put_Line("Suma = " & Add(3.0, 4.0)'Img );
51   Put_Line( "Max =" & Max(6.7, 8.9)'Img);
52   Put_Line( "Max2 =" & Max2(6.7, 8.9)'Img);
53   Print_Fibo(12);
54 end Lab1;

```

Pytanie

Jaka jest postać deklaracji funkcji i procedury ?

Funkcje i procedury można zdefiniować w momencie ich deklaracji.

- Procedury bez argumentów

```

1 procedure Lab1 is — procedura bez argumentów
2 — miejsce na deklaracje
3 begin
4 — ciało procedury
5 end Lab1;

```

- Procedury z argumentami

```

1 procedure Print_Fibo(N: Integer) is — procedura z argumentami
2 — więcej argumentów tworzymy po średniku
3 — miejsce na deklaracje
4 begin
5 — ciało procedury
6 end Print_Fibo

```

- Funkcja wyrażeniowa, jej ciało to pojedyncza instrukcja

```

1 function Add(A1, A2 : Float) return Float is
2   (A1 + A2);

```

- Zwykła funkcja z argumentami

```

1 function Max2(A1, A2 : in Float ) return Float is
2   — miejsce na deklaracje
3   begin
4     if A1 > A2 then return A1;
5     else return A2;
6   end if;
7 end Max2;

```

Zauważmy, że jeśli procedura lub funkcja nie ma wcale argumentów, to wówczas wcale nie piszemy nawiasów.

Pytanie

Co oznaczają słowa **in**, **out**, **in out** używane z argumentami procedur i funkcji ?

- **in** - parametr może być tylko czytany w ciele funkcji/procedury (defaultowo wszystkie argumenty są typu **in**)
- **out** - do parametru można zapisać dane, a po zapisaniu je także czytać
- **in out** - parametr może być zarówno czytany jak i nadpisywany

Przykłady

Listing 6: Błąd - nadpisywanie in

```

1 procedure Swap (A, B : Integer) is
2   Tmp : Integer;
3 begin
4   Tmp := A;
5
6   — Error: assignment to "in" mode parameter not allowed
7   A := B;
8   — Error: assignment to "in" mode parameter not allowed
9   B := Tmp;
10 end Swap;

```

Listing 7: Poprawna wersja

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure In_Out_Params is
4   — tutaj deklarujemy wszystkie funkcje, procedury i zmienne
5   — używane w procedurze głównej
6   procedure Swap (A, B : in out Integer) is
7     Tmp : Integer; — miejsce na deklaracje !
8   begin
9     Tmp := A;
10    A := B;
11    B := Tmp;
12  end Swap;
13
14  A : Integer := 12;
15  B : Integer := 44;
16 begin
17   Swap (A, B);
18   Put_Line (Integer'Image (A)); — Prints 44
19 end In_Out_Params;

```

Zwróćmy w powyższym przykładzie w jakich miejscach deklarowane są funkcje, procedury i zmienne oraz, że zmienne mogą być zadeklarowane *bez* lub z inicjalizacją, rolę operatora przypisania pełni w Adzie operator `:=`

Pamiętajmy też, że słowa kluczowe **in**, **out**, **in out** tyczą się tylko argumentów funkcji i procedur, a nie zmiennych, które zostały zadeklarowane przed ciałem procedur lub funkcji.

2.4. Rekordy

- Rekordy to złożony typ danych w Adzie
- Są odpowiednikiem klas z języków obiektowych
- Można zadeklarować domyślne wartości dla pól rekordu, wtedy jeśli zadeklarujemy zmienną typu tego rekordu i nie przypiszemy po nim wartości, zostaną im nadane wartości domyślne

Listing 8: Przykład rekordu w Adzie

```
1 type Date is record
2   — The following declarations are components of the record
3   Day   : Integer range 1 .. 31;
4   Month : Month_Type;
5   Year  : Integer range 1 .. 3000; — You can add custom constraints on fields
6 end record;
```

- Rekordom nadajemy wartość korzystając z tzw. agregatów, agregat to po prostu wartość (literał) dla rekordu (lub dowolnego innego typu złożonego, np. tablicy), wartości dla kolejnych pól zapisujemy po przecinku w nawiasach w przypadku rekordu

Listing 9: Przykłady agregatów dla typu Date

```
1 Ada_Birthday : Date := (10, December, 1815); — tutaj określamy wartości przez
   pozycje pola
2 Leap_Day_2020 : Date := (Day => 29, Month => February, Year => 2020); — tutaj
   określamy wartości podając nazwę pola
3 — ^ By name
```

- Do pól komponentu odwołujemy się podobnie jak do atrybutów obiektów w Javie/C++, podajemy nazwę pola po kropce

Listing 10: Stosowanie rekordów

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Record_Selection is
4
5   type Month_Type is
6     (January, February, March, April, May, June, July,
7      August, September, October, November, December);
8
9   type Date is record
10    Day   : Integer range 1 .. 31;
11    Month : Month_Type;
12    Year  : Integer range 1 .. 3000 := 2032;
13 end record;
14
15 Some_Day : Date := (1, January, 2000);
```

```

16
17 begin
18   Some_Day.Year := 2001;
19   Put_Line ("Day:" & Integer'Image (Some_Day.Day)
20           & ", Month:" & Month_Type'Image (Some_Day.Month)
21           & ", Year:" & Integer'Image (Some_Day.Year));
22 end Record_Selection;

```

Pytanie

Omów rekordy w Adzie.

2.5. Tablice

Tablice w Adzie są tym co rozumiemy przez tablicę w innych językach programowania - przechowują wiele zmiennych tego samego typu w jednym kontenerze. Mimo to, tablice w Adzie różnią się trochę sposobem w jaki je definiujemy i używamy, omówimy sobie te różnice.

- Tablicę w Adzie definiujemy tworząc nowy typ i określając, że typ ten ma być tablicą
- Przy definiowaniu tego typu, określamy jak będziemy indeksować tablicę, tutaj Ada jest inna od innych języków, indeksami tablicy mogą być dowolne wartości z dowolnego zakresu, zamiast podawać rozmiar tablicy, podajemy zakres indeksów który określa ten rozmiar
- Na końcu definicji musimy określić jakie typy ma przechowywać tablica

Listing 11: Definiowanie tablicy z jawnie określonymi indeksami

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 — Definiujemy tablice ktora bedzie indeksowana liczbami calkowitymi od 1 do 5
4 — Tablica ta bedzie miala 5 elementow, kazdy z nich bedzie typu My_Int
5 procedure Greet is
6   type My_Int is range 0 .. 1000;
7   type Index is range 1 .. 5;
8
9   type My_Int_Array is array (Index) of My_Int;
10  —                                     ^ Type of elements
11  —                                     ^ Bounds of the array
12  Arr : My_Int_Array := (2, 3, 5, 7, 11);
13  —                                     ^ Array literal, called aggregate in Ada
14 begin
15   for I in Index loop — Zwrocmy uwage na sposob iteracji !!!
16     Put (My_Int'Image (Arr (I))); — UWAGA ! Tak odwołujemy sie do elementow tablicy!
17     —                                     ^ Take the Ith element
18   end loop;
19   New_Line;
20 end Greet;

```

- indeksem tablicy może być dowolny typ, nie musi być to Integer
- nie jest konieczne deklarowanie osobnego typu dla indeksów tablicy, indeksy można równie dobrze zdefiniować przy definiowaniu tablicy

Listing 12: Różne typy jako indeksy tablicy

```

1 — różne typy mogą być indeksami:
2 type Month_Duration is range 1 .. 31;
3   type Month is (Jan, Feb, Mar, Apr, May, Jun,
4                 Jul, Aug, Sep, Oct, Nov, Dec);
5
6   type My_Int_Array is array (Month) of Month_Duration;
7
8 — ...
9 — Można też tak:
10 type My_Int is range 0 .. 1000;
11   type My_Int_Array is array (1 .. 5) of My_Int;

```

- w Laboratorium 1 mówiliśmy już o atrybutach (np. 'Image, 'Value), tablice też mają swoje atrybuty, są one szczególnie przydatne do iterowania po tablicy
- Atrybut 'Range pozwala na uzyskanie zakresu indeksów tablicy
- Atrybuty 'First i 'Last pozwalają na uzyskanie pierwszego i ostatniego indeksu tablicy
- Atrybut 'Length daje dostęp do długości tablicy

Listing 13: Atrybuty tablic

```

1 — atrybut 'Range
2 for I in Tab'Range loop
3   — ^ Gets the range of Tab
4   Put (My_Int'Image (Tab (I)));
5 end loop;
6
7 — ...
8 — atrybuty 'First i 'Last
9 for I in Tab'First .. Tab'Last - 1 loop
10  — ^ Iterate on every index except the last
11  Put (My_Int'Image (Tab (I)));
12 end loop;

```

- Jedną z najpotężniejszych cech tablic w Adzie są tablice bez zdefiniowanego zakresu indeksów
- Ada pozwala na utworzeniu typu tablicowego bez określenia zakresu indeksów, zakres określany jest dla konkretnej instancji, przy deklaracji typu określamy tylko typ jaki będą miały indeksy

Listing 14: Tablice bez określonych ograniczeń

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Unconstrained_Array_Example is
4   type Days is (Monday, Tuesday, Wednesday,
5               Thursday, Friday, Saturday, Sunday);
6
7   type Workload_Type is array (Days range <>) of Natural;
8   — Indefinite array type
9   — ^ Bounds are of type Days,
10  —   but not known
11
12   Workload : constant Workload_Type (Monday .. Friday) :=
13   — ^ Specify the bounds
14   —   when declaring

```

```

15 (Friday => 7, others => 8): — UWAGA: tak można !
16 — ^ Default value
17 — ^ Specify element by name of index
18 begin
19   for I in Workload'Range loop
20     Put_Line (Integer'Image (Workload (I)));
21   end loop;
22 end Unconstrained_Array_Example;

```

Pytanie

Omów tablice w Adzie.

2.6. Kwantyfikatory

- Kwantyfikatory to w Adzie konstrukcje językowe, które naśladują kwantyfikatory znane z matematyki - 'dla każdego', 'istnieje'
- Kwantyfikatory zwracają wartość typu *Boolean*
- Kwantyfikator sprawdza dla każdego elementu tablicy czy dany warunek (predykat określony przez programistę) jest spełniony

Listing 15: Przykład kwantyfikatorów 'dla każdego'

```

1 — ogólna forma kwantyfikatora 'dla każdego' to:
2 — (forall Identyfikator in [reverse])
3 — Definicja_Podtypu_Dyskretnego => Wyrażenie_Logiczne)
4 (forall I in A'First..(A'Last-1) => A(I)>=A(I+1) )
5 — lub dla kontenerów, tablic itp:
6 — (forall Identyfikator of [reverse] Kontener_lub_Tablica =>
7 — Wyrażenie_Logiczne)
8 (forall E of A => E>0.0 )

```

Listing 16: Przykład kwantyfikatorów 'istnieje'

```

1 — ogólna forma kwantyfikatora 'istnieje' to:
2 — (forsome Identyfikator in [reverse])
3 — Definicja_Podtypu_Dyskretnego => Wyrażenie_Logiczne)
4 (forsome I in A'Range => A(I) mod 2 =0)
5 — lub dla kontenerów, tablic itp:
6 — (forsome Identyfikator of [reverse])
7 — Kontener_lub_Tablica => Wyrażenie_Logiczne)
8 (forsome E of A => E=0.0 )

```

Pytanie

Omów kwantyfikatory w Adzie.

2.7. Instrukcje sterujące

- Instrukcja *if-then-else-end if* działa tak samo jak w innych językach programowania, po *if* następuje musi wyrażenie, które daje w wyniku wartość logiczną. *else* jest opcjonalne

Listing 17: if-then-else-end if

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4 procedure Check_Positive is
5   N : Integer;
6 begin
7   Put ("Enter an integer value: "); — Put a String
8   Get (N); — Reads in an integer value
9   Put (N); — Put an Integer
10  if N > 0 then
11    Put_Line (" is a positive number");
12  else
13    Put_Line (" is not a positive number");
14  end if;
15 end Check_Positive;

```

- Instrukcja *if-then-elsif-...-elsif-else-end if* pozwala na zapisanie kilku warunków po sobie

Listing 18: if-then-elsif-...-elsif-else-end if

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
3
4 procedure Check_Direction is
5   N : Integer;
6 begin
7   Put ("Enter an integer value: "); — Puts a String
8   Get (N); — Reads an Integer
9   Put (N); — Puts an Integer
10  if N = 0 or N = 360 then
11    Put_Line (" is due east");
12  elsif N in 1 .. 89 then — zwrocmy uwage na sprawdzanie zakresem !
13    Put_Line (" is in the northeast quadrant");
14  elsif N = 90 then — uzywamy jednego znaku = do porownywania !
15    Put_Line (" is due north");
16  elsif N in 91 .. 179 then
17    Put_Line (" is in the northwest quadrant");
18  elsif N = 180 then
19    Put_Line (" is due west");
20  elsif N in 181 .. 269 then
21    Put_Line (" is in the southwest quadrant");
22  elsif N = 270 then
23    Put_Line (" is due south");
24  elsif N in 271 .. 359 then
25    Put_Line (" is in the southeast quadrant");
26  else
27    Put_Line (" is not in the range 0..360");
28  end if;
29 end Check_Direction;

```

- pętla *for* tak jak w innych językach pozwala na wielokrotne wykonywanie tego samego kodu, liczba iteracji określana jest poprzez zakres

Listing 19: Podstawowa petla for

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5a is
4 begin
5   for I in 1 .. 5 loop

```



```

6      Put_Line ("Hello , World!" & Integer'Image (I)); — Procedure call
7      —      ^ Procedure parameter
8  end loop;
9 end Greet_5a;

```

- *reverse for* pozwala na iterowanie po indeksach w kolejności odwrotnej niż określona

Listing 20: reverse for

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5a_Reverse is
4 begin
5   for I in reverse 1 .. 5 loop
6     Put_Line ("Hello , World!" & Integer'Image (I));
7   end loop;
8 end Greet_5a_Reverse;

```

- pętla *while*

Listing 21: while

```

1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Greet_5c is
4   I : Integer := 1;
5 begin
6   — Condition must be a Boolean value (no Integers).
7   — Operator "<=" returns a Boolean
8   while I <= 5 loop
9     Put_Line ("Hello , World!" & Integer'Image (I));
10
11     I := I + 1;
12   end loop;
13 end Greet_5c;

```

Pytanie

Omów instrukcje sterujące w Adzie.

2.8. Definicje i zastosowania pojęć jak: atrybuty, agregaty, aspekty, pragmy, wyróżniki

2.8.1. Atrybuty

Atrybuty są częścią typów, obiektów, podprogramów i pozwalają na uzyskanie informacji lub wykonanie operacji na danym obiekcie. Do atrybutów odwołujemy się umieszczając po nazwie zmiennej pojedynczy apostrof, a po nim nazwę atrybutu.

Najczęściej używane atrybuty to *'Image* oraz *'Value*, które pozwalają na konwersję zmiennych różnego typu na typ *String* i odwrotnie:

Listing 22: 'Value i 'Image

```

1 declare
2   A: Integer := 99;
3 begin

```

```

4 Put_Line(Integer'Image(A));
5 A := Integer'Value("99");
6 end;

```

Atrybutów można również używać z zadeklarowanymi zmiennymi. Można także pisać w skrócie *Img* zamiast *Image*.

Pytanie

Czym są i do czego służą atrybuty ?

2.8.2. Agregaty

Patrz -> Rekordy

2.8.3. Pragmy, Aspekty

Generalnie to instrukcje dla kompilatora, nie będziemy ich tu jednak zgłębiać.

2.9. Wskaźniki i ich rodzaje. Element listy. Wskaźniki do podprogramów.

W Adzie mamy dwa rodzaje wskaźników

- ograniczony - do obsługi dynamicznych struktur danych, np. list, drzew, grafów

```

1 type Nazwa_Typu_Wskaznikowego is access Wskazanie_Podtypu;
2 % np.
3 type Wsk_Int is access Integer;

```

- ogólny - wskazują na zadeklarowane obiekty lub podprogramy

```

1 type Nazwa_Typu is access constant Typ;
2 % np.
3 type Wsk_St_Integer is access constant Integer;

```

- W Adzie możliwe jest korzystanie ze wskaźników do obiektów, które są tym czym są wskaźniki w innych językach programowania, np. C++
- Aby móc utworzyć wskaźnik do danego typu, musimy utworzyć nowy typ, który będzie typu *Access*, a następnie podajemy do jakiego typu ma on być wskaźnikiem
- *Access* to ogólny typ, który sygnalizuje, że zmienna jest wskaźnikiem, następnie podajemy do jakiego typu ma ona być wskaźnikiem
- *Access* (wskaźnik) może zostać zdefiniowany w kilku trybach (podobnie jak argument do funkcji), tryby te różnią się tym czy wskaźnik może być modyfikowany czy nie
- Tryb *all* to tryb, który pozwala zarówno na odczyt jak i zapis do zmiennej wskaźnikowej

Pytanie

Jakie są rodzaje wskaźników w Adzie i jak je zapisujemy w kodzie ? Czym są tryby wskaźników w Adzie i jak je oznaczamy w kodzie ?

Przykład

Listing 23: Tworzenie wskaźników w Adzie

```

1 — definiujemy nowy typ który jest obiektem
2 type Element is
3   record
4     Data : Integer := 0;
5     Next : access Element := Null;
6   end record;
7
8 — definiujemy nowy typ, który jest wskaźnikiem do naszego typu
9 — wskaźnik ten jest zdefiniowany w trybie all
10 type Elem_Ptr is access all Element;

```

2.9.1. new keyword

- W Adzie możliwe jest tworzenie obiektów recordów przy użyciu słowa kluczowego **new**
- Taka instrukcja zwraca wskaźnik do nowo utworzonego obiektu i może zostać przypisana do zmiennej wskaźnikowej

Przykład

Listing 24: Użycie słowa new

```

1 with Dates; use Dates;
2
3 package Access_Types is
4   type Date_Acc is access Date;
5
6   D : Date_Acc := new Date;
7   —           ^ Allocate a new Date record
8 end Access_Types;

```

- Możliwe jest też zainicjalizowanie obiektu przy jego tworzeniu

Listing 25: Użycie słowa new

```

1 with Dates; use Dates;
2
3 package Access_Types is
4   type Date_Acc is access Date;
5   type String_Acc is access String;
6
7   D : Date_Acc := new Date'(30, November, 2011);
8   Msg : String_Acc := new String'("Hello");
9 end Access_Types;

```

2.9.2. Dereferencja wskaźników

- Dereferencje całego obiektu można uzyskać stosując składnię *.all*
- Jeśli chcemy uzyskać dostęp do jednego z atrybutów wystarczy zapisać jego nazwę po kropce

Przykład

Listing 26: Dereferencja

```

1 with Dates; use Dates;
2
3 package Access_Types is
4   type Date_Acc is access Date;
5
6   D      : Date_Acc := new Date'(30, November, 2011);
7
8   Today : Date := D.all;
9   —      ^ Access value dereference
10  J      : Integer := D.Day;
11  —      ^ Implicit dereference for record and array components
12  —      Equivalent to D.all.day
13 end Access_Types;

```

2.10. Pakiety

- Pomimo, że Ada pozwala na definiowanie wszystkich funkcji, procedur i typów w jednym pliku, nie jest to oczywiście dobre rozwiązanie gdy mamy napisać większą aplikację
- Podział programów na pakiety w Adzie jest tak prosty, że aż zęby bołą
- Nowy pakiet definiujemy przy użyciu słowa `package` na początku pliku
- Pakiet importujemy przy użyciu słowa `with` na początku pliku

Listing 27: Definicja pakietu - plik week.ads

```

1 package Week is — Zwroc uwage na rozszerzenie pliku ! Potem to wyjasnimy !
2
3   Mon : constant String := "Monday";
4   Tue : constant String := "Tuesday";
5   Wed : constant String := "Wednesday";
6   Thu : constant String := "Thursday";
7   Fri : constant String := "Friday";
8   Sat : constant String := "Saturday";
9   Sun : constant String := "Sunday";
10
11 end Week;

```

Listing 28: Uzycie pakietu

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Week;
3 — References the Week package, and adds a dependency from Main
4 — to Week
5
6 procedure Main is
7 begin
8   Put_Line ("First day of the week is " & Week.Mon);
9 end Main;

```

- korzystając z `use` można używać zmiennych i funkcji bez nazw kwalifikowanych
- Plik `.ads` służy tylko do tworzenia deklaracji, definicje tych deklaracji umieszczamy w pliku o takiej samej nazwie, ale z rozszerzeniem normalnym dla Ady, czyli `.adb`

- W pliku w który znajdują się definicje używamy słowa kluczowego *body* na początku pliku, patrz przykłady poniżej

Listing 29: operations.ads

```

1 package Operations is
2
3   — Declaration
4   function Increment_By
5     (I : Integer;
6      Incr : Integer := 0) return Integer;
7
8   function Get_Increment_Value return Integer;
9
10 end Operations;
```

Listing 30: operations.adb

```

1 package body Operations is — UWAGA: musi być body !
2
3   Last_Increment : Integer := 1; — Do tej zmiennej nie ma dostępu program
4   — który importuje ten pakiet !!!
5   function Increment_By
6     (I : Integer;
7      Incr : Integer := 0) return Integer is
8   begin
9     if Incr /= 0 then
10      Last_Increment := Incr;
11    end if;
12
13    return I + Last_Increment;
14  end Increment_By;
15
16  function Get_Increment_Value return Integer is
17  begin
18    return Last_Increment;
19  end Get_Increment_Value;
20
21 end Operations;
```

Listing 31: Użycie pakietu operations

```

1 with Ada.Text_IO; use Ada.Text_IO;
2 with Operations;
3
4 procedure Main is
5   use Operations;
6
7   I : Integer := 0;
8   R : Integer;
9
10  procedure Display_Update_Values is
11    Incr : constant Integer := Get_Increment_Value;
12  begin
13    Put_Line (Integer'Image (I)
14              & " incremented by " & Integer'Image (Incr)
15              & " is " & Integer'Image (R));
16    I := R;
17  end Display_Update_Values;
18 begin
19   R := Increment_By (I);
```

```
20 Display_Update_Values;  
21 R := Increment_By (1);  
22 Display_Update_Values;  
23  
24 R := Increment_By (1, 5);  
25 Display_Update_Values;  
26 R := Increment_By (1);  
27 Display_Update_Values;  
28  
29 R := Increment_By (1, 10);  
30 Display_Update_Values;  
31 R := Increment_By (1);  
32 Display_Update_Values;  
33 end Main;
```

- Jeśli chcemy skompilować program składający się z wielu modułów korzystamy z narzędzia *gnat-make*
- Jako argument dla tej komendy przekazujemy nazwę pliku z plikiem, który zawiera główną procedurę
- Narzędzie samo znajduje wszystkie potrzebne dependencje, kompiluje je i linkuje więc w wyniku dostajemy wykonywalny program o nazwie takiej jak plik z rozszerzeniem .adb

Pytanie

Omów pakiety w Adzie.

2.11. Wyjątki

W Adzie mamy dwa rodzaje wyjątków:

- wyjątki predefiniowane - udostępniane przez język w swoich pakietach
- wyjątki użytkownika - zadeklarowane przez programistę

Wyjątki zgłasza się przez użycie słowa kluczowego *raise*:

```
1 raise [Nazwa_Wyjatku] [with Komunikat];  
2 raise Error;  
3 raise Pewien_Blad with "Komunikat o wyjątku";
```

Wyjątki obsługujemy w segmencie kodu o etykiecie *exception*:

```
1 procedure Proc is  
2 begin  
3 — treść procedury  
4 exception  
5 when others => Put_Line("Błąd w procedurze Proc");  
6 end Proc;
```

Pytanie

Omów wyjątki w Adzie.

2.12. Zadania i typy zadaniowe. Tworzenie i kończenie zadań.

Zadania są tym w Adzie, co w Javie określamy wątkiem. Zadania definiujemy w programie z użyciem słowa kluczowego Task.

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Simple_Task is
4   task T;
5
6   task body T is
7   begin
8     Put_Line ("In task T");
9   end T;
10 begin
11   Put_Line ("In main");
12 end Show_Simple_Task;
```

Wątek jest uruchamiany w momencie uruchamiania procedury głównej programu, nie wymaga specjalnego uruchamiania w kodzie. Zadania kończy się gdy wykona wszystkie swoje instrukcje lub gdy zgłosi wyjątek lub gdy wykona się instrukcja terminate lub abort.

Komunikacja w zadaniach odbywa się przez tzw. mechanizm *rendez-vous*. Task definiuje nazwę entry, a następnie w swoim ciele akceptuje tę nazwę, czyli czeka aż inny proces wywoła nazwę tego entry. Przykład

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Rendezvous is
4
5   task T is
6     entry Start;
7   end T;
8
9   task body T is
10  begin
11    accept Start; — Waiting for somebody to call the entry
12    Put_Line ("In T");
13  end T;
14
15 begin
16   Put_Line ("In Main");
17   T.Start; — Calling T's entry
18 end Show_Rendezvous;
```

Można też zdefiniować wybór pomiędzy entries z użyciem pętli i select:

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure Show_Rendezvous_Loop is
4
5   task T is
6     entry Reset;
7     entry Increment;
8   end T;
9
10  task body T is
11    Cnt : Integer := 0;
12  begin
13    loop
14      select
15        accept Reset do
16          Cnt := 0;
17        end Reset;
```

```

18     Put_Line ("Reset");
19   or
20     accept Increment do
21       Cnt := Cnt + 1;
22     end Increment;
23     Put_Line ("In T's loop (" & Integer'Image (Cnt) & ")");
24   or
25     terminate;
26   end select;
27   end loop;
28   end T;
29
30 begin
31   Put_Line ("In Main");
32
33   for I in 1 .. 4 loop
34     T.Increment; — Calling T's entry multiple times
35   end loop;
36
37   T.Reset;
38   for I in 1 .. 4 loop
39     T.Increment; — Calling T's entry multiple times
40   end loop;
41
42 end Show_Rendezvous_Loop;

```

Można też zastosować instrukcję select z dozorami:

```

1 select
2   when Warunek1 =>
3     accept We1 do
4       — instrukcje
5     end We1;
6     — instrukcje
7   or
8   when Warunek2 =>
9     accept We2 do
10      — instrukcje
11    end We2;
12    — instrukcje
13 end select;

```

Pytanie

Jak realizowana jest wielowątkowość w Adzie ?

2.13. Zmienne dzielone

To zmienne, które wykorzystywane są przez dwa lub więcej zadań. Bezpieczny dostęp do takiej zmiennej zapewniamy w Adzie dodając do niej pragnę *atomic*.

```

1 procedure Zadania is
2   Koniec: Boolean := False
3   with Atomic;
4
5 task Zadanie1;
6 task Zadanie2;
7 task body Zadanie1 is
8   begin
9     loop

```



```
10     exit when Koniec;  
11     ...  
12     end loop;  
13 end Zadanie1;  
14  
15 task body Zadanie2 is  
16     begin  
17         loop  
18             exit when Koniec;  
19             ...  
20         end loop;  
21 end Zadanie2;  
22  
23 begin — procedury  
24 ...  
25 Koniec := True;  
26 end Zadania;
```

Pytanie

Omów realizację zmiennych dzielonych w Adzie.

2.14. Obiekt chroniony

Obiekt chroniony jest konstrukcją języka Ada. Bazuje na dwóch pojęciach:

- sekcja krytyczna
- warunkowy rejon krytyczny

Obiekt chroniony zawiera w sobie operacje i dane. Tylko do operacji jest dostęp z zewnątrz. Operacje wykonują się z wzajemnym wykluczaniem. Obiekty chronione używane są do asynchronicznej komunikacji. W Adzie obiekty chronione oznaczamy słowem kluczowym *protected*.

Pytanie

Omów obiekty chronione w Adzie.

3 Wiedza ogólna

3.1. Proces i jego stany. Szeregowanie. Przeplot. Priorytety.

- Program definiujemy jako ciąg instrukcji dla maszyny (pojęcie statyczne). Natomiast proces możemy utożsamiać z programem, który jest wykonywany na danej maszynie (pojęcie dynamiczne). Proces ma swój własny oddzielny obszar pamięci niezależny od innych procesów.
- Obsługą procesów zajmuje się program szeregujący (część systemu operacyjnego)
- Możliwe stany procesu (wersja minimalna):
 - gotowy - czeka na przydział procesora
 - aktywny - wykonywany przez procesor
 - zawieszony - oczekuje na spełnienie swoich żądań

Pytanie

Opisz czym jest proces i jakie może mieć stany.

Szeregowanie polega na przydziale procesora do procesów. Tą czynnością zajmuje się program systemu operacyjnego nazywany scheduler. Program ten przydziela procesy do procesora według pewnego przyjętego algorytmu.

Szeregowanie z wywłaszczaniem to takie w którym scheduler decyduje kiedy dany proces zostanie przełączony, a bez wywłaszczania to sytuacja w której to proces informuje system operacyjny, że chce uzyskać dostęp do procesora. Większość współczesnych systemów opiera się o szeregowanie z wywłaszczaniem.

Pytanie

Opisz czym jest szeregowanie. Jakie znasz dwa rodzaje ?

Przeplot to technika umożliwiająca wirtualną realizację równoległości. Jeśli mamy do dyspozycji tylko jeden procesor, to równoległe wykonywanie na nim wielu procesów nie jest możliwe, dlatego stosuje się szybkie zmiany wykonywanego procesu na procesorze, czyli właśnie tzw. przeplot.

Pytanie

Opisz czym jest przeplot.

Priorytety określają względną ważność procesu. Priorytet wykorzystywany jest w algorytmie szeregowania przez system operacyjny. Priorytet może być:

- statyczny - wartość nadana początkowo nie zmienia się
- dynamiczny - wartość priorytetu może zmieniać się w trakcie wykonywania procesu

Pytanie

Opisz czym są priorytety i jakie znasz ich rodzaje.

3.2. Współbieżność a równoległość.

Jaka jest różnica ?

- równoległość - oznacza wykonywanie wielu procesów jednocześnie w tym samym czasie
- współbieżność - oznacza, że jeden proces rozpoczyna się przed zakończeniem drugiego, ale nie są one wykonywane jednocześnie, takie podejście wymaga przeplotu
- (bonus) przetwarzanie rozproszone - informacja jest obrabiana jednocześnie przez wiele komputerów (procesorów), rozmieszczonych terytorialnie i połączonych ze sobą w sieć. Wykonują one osobno poszczególne etapy zadania i odsyłają wyniki do jednego wspólnego centrum nadzoru.

Pytanie

Omów różnicę pomiędzy współbieżnością, a równoległością.

3.3. Wzajemne wykluczanie. Sekcja krytyczna.

Sekcja krytyczna to niepodzielny ciąg instrukcji, wykonywany w tym samym czasie tylko przez jeden proces czyli z wzajemnym wykluczaniem.

Pytanie

Omów czym jest sekcja krytyczna i wzajemne wykluczanie.

3.4. Metody synchronizacji i komunikacji procesów, w tym semaforey, komunikaty, monitory, spotkania.

Zacznijmy od tego czym w ogóle jest synchronizacja i komunikacja procesów:

- synchronizacja - zapewnienie wymagań kolejnościowych, proces, który ma coś zrobić wcześniej zrobi to wcześniej
- komunikacja - przesyłanie danych pomiędzy procesami, komunikacja może być synchroniczna lub asynchroniczna

Omówimy sobie teraz kilka mechanizmów służących do synchronizacji i komunikacji procesów.

- Zmienne dzielone - to jedna z najprostszych metod komunikacji procesów, polega na dostępie do tej samej zmiennej przez kilka procesów, operacje wykonywane na tej zmiennej powinny być atomowe
- Semaforey - semafor to zazwyczaj liczba całkowita o wartościach nieujemnych, która zmienia swój stan w zależności od tego czy dana sekcja programu jest wykonywana przez jakiś proces, semafor ogranicza liczbę procesów, które mogą wykonywać daną część programu, inne procesy muszą czekać aż zmienna zmieni swój stan na taki, który pozwoli na dostęp
- Monitor - obiekt, zawierający wykluczające się procedury, tzn. tylko jedna z procedur może być wykonywana w danym momencie czasu

Spotkania omówimy sobie w następnym podrozdziale.

Pytanie

Omów podstawowe metody komunikacji i synchronizacji procesów.

3.5. Spotkania i ich rodzaje.

Mechanizm synchronizacyjny udostępniany przez Adę to tzw. spotkania albo randki. W spotkaniu uczestniczą dwa (lub więcej) procesy, które w Adzie noszą nazwę zadań. Zaczniemy od omówienia randki między dwoma zadaniami.

Spośród zadań uczestniczących w randce jedno jest zadaniem aktywnym, a drugie zadaniem pasywnym. Zadanie aktywne inicjuje randkę, ale w czasie jej trwania nie robi nic. Zadanie pasywne jest zadaniem, które udostępnia pewne wejścia. Wejścia te mogą być wywoływane przez zadanie aktywne, które chce zainicjować randkę. Gdy dojdzie do randki, zadanie pasywne zajmuje się jej obsługą, wykonując określony fragment programu, podczas gdy zadanie aktywne jest wstrzymywane w oczekiwaniu na zakończenie randki.

Zadania w Adzie składają się z dwóch części. Pierwsza część to tzw. specyfikacja zadania. Specyfikacja określa jakie wejścia są udostępniane przez zadanie i jakie są ich argumenty. Właściwa treść zadania jest określona w osobnym fragmencie kodu.

Spotkania w Adzie mogą być symetryczne lub asymetryczne.

Pytanie

Omów spotkania i ich rodzaje.

3.6. Komunikacja synchroniczna i asynchroniczna.

- komunikacja synchroniczna - z komunikacją synchroniczną mamy do czynienia wtedy, gdy chcąc się ze sobą skomunikować procesy są wstrzymywane do chwili, gdy komunikacja będzie się mogła odbyć.
- komunikacja asynchroniczna - komunikacja asynchroniczna nie wymaga współlistnienia komunikujących się procesów w tym samym czasie. Polega na tym, że nadawca wysyła komunikat nie czekając na nic. Komunikaty są buforowane w jakimś miejscu (odpowiada za to system operacyjny lub mechanizmy obsługi sieci) i stamtąd pobierane przez odbiorcę.

Pytanie

Omów różnicę pomiędzy komunikacją synchroniczną, a asynchroniczną.

3.7. Proces/wątek - podobieństwa i różnice.

Różnice:

- Wątki zostały zaprojektowane aby komunikować się ze sobą, a procesy działają w większości wypadków niezależnie
- Każdy proces ma własną przestrzeń pamięci, a wątki korzystają z zasobów procesu
- Kiedy kończy się proces umierają wszystkie wątki, ale koniec wątku nie oznacza końca procesu
- Wątki zawierają się w procesach

Podobieństwa:

- Zarówno wątki jak i procesy mają swoje id
- Wątki i procesy mają priorytety
- Możemy tworzyć procesy wewnątrz procesów, tak jak wątki wewnątrz wątków

Pytanie

Omów podobieństwa i różnice między wątkami, a procesami.