7. előadás

Paraméterátadás. Paraméterek az Adában. Túlterhelés. Felüldefiniálás. Kifejezések.

Paraméterek

- Paraméterezhető dolgok
  - Alprogramok
  - Típusok (diszkrimináns, indexhatár)
  - Sablonok
- Formális paraméter - aktuális paraméter
- Paraméterek megfeleltetése
- Alprogram: paraméterátadás

Alprogram-paraméterek

- Értékek és objektumok
- Bemenő és kimenő paraméter
  - Az információáramlás iránya
  - Bemenő paraméter: jobbérték
  - Kimenő paraméter: balérték
- Paraméterátadási módok
  - Technikai, nyelvimplementációs kérdés

Paraméterátadási módok

- Szövegszerű helyettesítés (makróknál)
- Név szerinti (archaikus, Algol 60, Simula 67)
- Érték szerinti (C, Pascal)
- Cím szerinti (Pascal, C++)
- Eredmény szerinti (Ada)
- Érték/eredmény szerinti (Algol W)
- Megosztás szerinti (Java, Eiffel, CLU)
- Igény szerinti (lusta kiértékelésű funkcionális ny.)

Szövegszerű helyettesítés

- A legelső programozási nyelvek assembly-k voltak, abban makrókat lehetett írni
- A makrók még mindig fontosak: pl. C, C++
- A makró törzsében a formális paraméter helyére beíródik az aktuális paraméter szövege
- Egyszerű működési elv, de szemantikusan bonyolult: könnyű hibát véteni
- A makró törzse beíródik a hívás helyére

C makró

```
#define  square(x)   x*x
int x = square(3+7);
```

3+7*3+7

```
#define  max(a,b)   ((a)>(b) ? (a) : (b))
int   x = 5,   y = max(++x, 3);
```

x ← 7

- Egyéb furcsaságokat okoz az, hogy a makró törzse a hívás helyére behelyettesítődik
- Nem lehet swap makrót írni

Inline alprogram

- A makró „hatékonyabb”, mint az alprogram
  - kisebb futási idő
  - nagyobb programkód
- Ugyanezt tudja a kifejtett (inline) alprogram
  - rendes paraéterátadás
  - szemantikus biztonság

Javaslat a fordítónak

function Max (A, B: Integer) return Integer…
pragma Inline(Max);

inline int max(int a, int b) { return a > b ? a : b; }

Optimalizálás

- Egy jó fordító esetleg jobban optimalizál, mint a programozó
- Inline kifejtés
- Tár vagy végrehajtási idő?
- Ada: Optimize fordítóvezérlő direktíva

```
pragma Optimize(Time);     -- Space, Off
```

Vissza: paraméterátadási módok

- Érték szerinti
- Cím szerinti
- Eredmény szerinti
- Érték/eredmény szerinti

- Ezeket kötelező ismerni, érteni

Érték szerinti paraméterátadás

- Call-by-value
- Nagyon elterjedt (C, C++, Pascal, Ada)
- Bemenő szemantikájú
- A formális paraméter az alprogram lokális változója
- Híváskor az aktuális értéke bemásolódik a formálisba
- A vermen készül egy másolat az aktuálisról
- Kilépéskor a formális megszűnik

Érték szerint a C++ nyelvben

```
int lnko ( int a, int b )
{
        while (a != b)
                if (a>b) a-=b; else b-=a;
}
int f ()
{
        int x = 10, y = 45;
        return lnko(x,y) + lnko(4,8);
}
```

x és y nem változik

hívható bal- és jobbértékkel

Cím szerinti paraméterátadás

- Call-by-reference
- A Fortran óta széles körben használt
- Kimenő szemantikájú
  - pontosabban be- és kimenő
  - csak balértékkel hívható
- A formális paraméter egy címet jelent
- A híváskor az aktuális paraméter címe adódik át
- A formális és az aktuális paraméter ugyanazt az objektumot jelentik (alias)

Cím szerint a C++ nyelvben

```
void swap ( int& a, int& b )
{
        int c = a;
        a = b;   b = c;
}
void f ()
{
        int x = 10, y = 45;
        swap ( x, y );
        // értelmetlen:   swap ( 3, 4 );
}
```

x és y megváltozik

Pascal-szerű nyelvek

```
int lnko ( int a, int b ) …
void swap ( int& a, int& b ) …

function lnko ( a, b: integer ) : integer …
procedure swap ( var a, b: integer ) …
```

Érték/eredmény szerinti paraméterátadás

- Call-by-value/result
- Algol-W, Ada (kis különbséggel)
- Kimenő szemantika
  - pontosabban be- és kimenő
  - csak balértékkel hívható
- A formális paraméter az alprogram lokális változója
- Híváskor az aktuális értéke bemásolódik a formálisba
- A vermen készül egy másolat az aktuálisról
- Kilépéskor a formális értéke visszamásolódik az aktuálisba

Érték/eredmény szerint az Adában

```
procedure Swap ( A, B: in out Integer ) is
        C: Integer := A;
begin
        A := B;  B := C;
end Swap;
procedure P is
        X: Integer := 10;
        Y: Integer := 45;
begin
        Swap ( X, Y );          -- a Swap(3,4) értelmetlen
end P;
```

X és Y megváltozik

Eredmény szerinti paraméterátadás

- Call-by-result
- Ada
- Kimenő szemantika
  - csak balértékkel hívható
- A formális paraméter az alprogram lokális változója
- Kilépéskor a formális értéke bemásolódik az aktuálisba
  - Híváskor az aktuális értéke nem másolódik be a formálisba

Eredmény szerint az Adában

```
procedure Betűt_Olvas ( C: out Character ) is
begin
        Ada.Text_IO.Get ( C );
        if C < ‘A’ or else C > ‘Z’ then
                raise Constraint_Error;
        end if;
end;
C: Character;
…
        Betűt_Olvas(C);
```

- C nem volt inicializálva
- C értéket kapott

Adatmozgatással járó paraméterátadás

- Data transfer
- Ilyen az érték, az eredmény és az érték/eredmény szerinti
  - Nem ilyen a cím szerinti
- Az aktuális paraméterről másolat készül
  - Független az aktuális a formálistól
  - Ha valamelyik változik, a másik nem
  - Könnyebben követhető

Cím versus érték/eredmény szerinti

- Mindkettő (be- és) kimenő szemantikájú
- Az utóbbi adatmozgatásos
  - nagy adat esetén a cím szerinti hatékonyabb
  - kis adat esetén az érték/eredmény szerinti hatékonyabb lehet (ha sok a hivatkozás)
- „Ugyanaz” a program másként működhet
- Az érték/eredmény szerinti általában jobban érthető viselkedést produkál
  - Mindkettőnél adható csúful viselkedő példa

Példa: egy Ada kódrészlet

```
N: Integer := 4;
procedure P ( X: in out Integer ) is
begin
        X := X + 1;
        X := X + N;
end P;
…
        P ( N );
```

Cím szerinti: 10

Érték/eredmény: 9

Paraméterátadási módok az Adában

- Bemenő (in) módú paraméterek:
  érték szerint vagy cím szerint
- Kimenő (out), illetve be- és kimenő (in out) módú paraméterek:
  - Bizonyos típusoknál **(érték/)eredmény szerint** (pl. skalár típusok, rekordok, mutatók)
    pass by copy
  - Más típusoknál **cím szerint**
    (pl. jelölt típusok, taszkok, védett egységek)
  - Egyes típusoknál implementációfüggő
    (pl. tömbök)

Információáramlás

- Az Ada programozó az információáramlással foglalkozik, nem a paraméterátadás részleteivel
- Szigorú statikus szabályok a formális paraméterek használatára
  - in módút csak olvasni szabad
  - out módút először inicializálni kell
    - Ada 83: egyáltalán nem olvasható
  - in out módú: nincs megkötés

Például:

```
procedure E (  Vi: in Integer;
               Vo: out Integer;
               Vio: in out Integer )
is
begin
   Vio := Vi + Vo;      -- helytelen, Vo-t nem olvashatjuk
   Vi := Vio;           -- helytelen, Vi-t nem írhatjuk
   Vo := Vi;            -- helyes
   Vo := 2*Vo+Vio;      -- helyes, Vo már kapott értéket
end E;
```

in módú paraméterek

- Akár érték, akár cím szerint történhet
- Mindenféleképpen bemenő szemantikájú
- Fordítási hiba, ha írni akarom
- A fordító dönthet, melyik a szerencsésebb az adott esetben
- Ha a programozó trükközne:
  - ha olyan kódot ír, ami függ a paraméterátadási módtól,
  - akkor hibás a program
    - „bounded error”, nem feltétlenül veszi észre a fordító vagy a futtató rendszer

A „bemenő” szemantika hangsúlyozása más nyelvekben

- Általában nem ilyen tiszta koncepció
- C++

```
int f ( const T& p )
{
    p = …     // fordítási hiba
}
```

bemenő, de cím szerinti

- Modula-3: READONLY paraméter

Történelem

- Makrók: szövegszerű behelyettesítés
- FORTRAN: csak „be- és kimenő” paraméter
  - régi változatok: cím, újabbak esetenként é/e
  - FORTRAN IV: hihetetlen, de literál is megváltozhat
- Algol 60: név és (kérésre) érték szerinti
  - Simula 67: érték és (kérésre) név szerinti
- COBOL, Algol 68, Pascal, C(++): érték és cím
- Algol W: érték és érték/eredmény
- Tisztán objektumelvű nyelvek: megosztás sz.
- Lusta kiértékelésű funkcionális nyelvek: igény sz.

Kommunikáció programegységek között

- Nonlokális és globális változók
  - Általában nem szerencsés, nem javasolt
  - Néha hasznos
    - Rövidebb a paraméterlista
    - Hatékonyabb lehet a kód
  - Blokkok, hatókör, blokkszerkezetes nyelvek
- Paraméterek

Alprogram nonlokális változói

```
procedure Rendez ( T: in out Tömb ) is
        function Max_Hely ( T: Tömb ) return Index is
                Mh: Index := T'First;
        begin
                for I in T'Range loop
                        if T(Mh) < T(I) then Mh := I; end if;
                end loop;
                return Mh;
        end;
begin
        for I in reverse T'Range loop
                Mh := Max_Hely( T(T'First .. I) );
                Felcserél( T(I), T(Mh) );
        end loop;
end Rendez;
```

Alprogram nonlokális változói

```
procedure Rendez ( T: in out Tömb ) is
        function Max_Hely ( Vége: Index ) return Index is
                Mh: Index := T'First;
        begin
                for I in T'First .. Vége loop
                        if T(Mh) < T(I) then Mh := I; end if;
                end loop;
                return Mh;
        end;
begin
        for I in reverse T'Range loop
                Mh := Max_Hely( I );
                Felcserél( T(I), T(Mh) );
        end loop;
end Rendez;
```

Paraméter alapértelmezett értéke

- Bizonyos nyelvekben (C++, Ada, …)
- A formális paraméter deklarációjában alapértelmezett érték adható meg
- Híváskor nem kötelező aktuálist megfeleltetni neki
- Csak bemenő szemantika esetén értelmes
  - C++ esetében: csak érték szerintinél

Példa az Adában és a C++-ban

```
void inc ( int& x, int d = 1 ) { x += d; }

procedure Inc ( X: in out Integer; D: in Integer := 1 ) is
begin
        X := X + D;
end Inc;

int n = 4;                                  N: Integer := 4;

inc( n, 3 );   inc( n );                    Inc( N, 3 );    Inc( N );
```

Paraméterek megfeleltetése

- A programozási nyelvek többségében: pozícionális (az i. formálisnak az i. aktuális)
- Az Adában:
  - pozícionális formában (positional association)
  - névvel jelölt formában (named association)
  - lehet keverni is a kettőt

Névvel jelölt paraméter-megfeleltetés

```
procedure Swap ( A, B: in out Integer ) is
        C: Integer := A;
begin
        A := B;  B := C;
end Swap;

Swap ( X, Y );                  -- pozícionális
Swap ( A => X,  B => Y );       -- névvel jelölt
Swap ( B => Y,  A => X );       -- a sorrend tetszőleges
Swap ( X,  B => Y );            -- keverni is lehet
```

Alapértelmezett érték + névvel jelölt forma

```
void f ( int x, int y, int w = 200, int h = 200, int c = 0 ) …
```

- f (5, 7) f (5, 7, 100) f (5, 7, 200, 200, 1)
- fontossági sorrendben

```
procedure F ( X, Y: in Integer;
                    W, H: Integer := 200; C: Integer := 0 ) …
```

- F (5, 7) F (5, 7, 100) F (5, 7, C => 1)

Egyéb paraméterezett dolgok (1)

- Sablon, diszkriminánsos típus
- Analógia a lehetőségekben
  - Alapértelmezett érték a bemenő paraméternek
  - Névvel jelölt paraméter-megfeleltetés
- A névvel jelölt megfeleltetést másnál is használjuk majd…

Egyéb paraméterezett dolgok (2)

```
generic
        type T is private;
        with function "+" ( A,B: T ) return T is <>;
        Z: in Integer := 0;
package P is
        type R ( D: Boolean := True ) is private;
    …
end P;
…
package I is new P ( Integer, Z => 1 );
X: I.R( D => False );
```

Alprogramnevek túlterhelése

- Overloading
- Ugyanazzal a névvel több alprogram
- Különböző legyen a szignatúra
- A fordító a hívásból eldönti, hogy melyiket kell meghívni
  - Ha egyik sem illeszkedik: fordítási hiba
  - Ha több is illeszkedik: fordítási hiba
- Ha ugyanazt a tevékenységet különböző paraméterezéssel is el akarjuk tudni végezni

Példák C++ és Ada nyelven

```
int max ( int x, int y ) {  return x > y ? x : y;  }
int max ( int x, int y, int z ) {  return max(x, max(y,z));  }
int x = max( 50*62, 51*60, 52*61 );

function Max ( X, Y: Integer ) return Integer is
    begin  if X > Y then return X; else return Y; end if; end;
function Max ( X, Y, Z: Integer ) return Integer is
    begin  return Max(X, Max(Y,Z)); end;
X: Integer := Max( 50*62, 51*60, 52*61 );
```

Szignatúra

- C++: a név és a formális paraméterek száma és típusa
- Ada: a név, a formális paraméterek száma és típusa, valamint a visszatérési típus
  - túl lehet terhelni a visszatérési értéken
  - nem lehet egy függvényt úgy hívni, hogy semmire sem használjuk a visszatérési értékét

Különböző szignatúra

```
procedure A;
procedure A ( I: in out Integer );
procedure A ( S: String );
function A return Integer;
function A return Float;


procedure A ( V: Natural := 42 );   -- nem jó
```

Híváskor a többértelműség feloldandó

```
package Igék is
        type Ige is ( Sétál, Siet, Vár );
        procedure Kiír ( I: in Ige );
end Igék;
package Főnevek is
        type Főnév is ( Ház, Palota, Vár );
        procedure Kiír ( F: in Főnév );
end Főnevek;
use Igék, Főnevek;
```

```
Kiír( Vár );
    -- ford. hiba


Igék.Kiír( Vár );
Kiír( Ige'(Vár) );
Kiír( I => Vár );
```

Operátorok túlterhelése

- Mind a C++, mind az Ada nyelvben lehet
  - C++-ban több operátor van (pl. () vagy =)
  - Az Adában nem minden operátort lehet túlterhelni (nem lehet: in, not in, and then, or else)
- Egyes nyelvekben csak közönséges alprogramokat (Java), vagy még azt sem (ML)
  - De a predefinit operátorok általában túlterheltek (+)
- Egyes nyelvekben lehet új operátorokat is definiálni (ML, Clean stb.)
  - fixitás, precedencia, asszociativitás megadásával

Operátorok túlterhelése Adában

```
function "*" ( A, B: Vektor ) return Real is
        S: Real := 0.0;
begin
        for I in A'Range loop
                S := S + A(I) * B(I);
        end loop;
        return S;
end "*";
```

R := P * Q; R := "*"(P,Q);

Alprogram, mint típusművelet

- Beépített típusok: operátorok, attribútumok
- Származtatott típusok: megöröklik ezeket

- Programozó által definiált típusok
  - Tipikusan: átlátszatlan típus
  - Absztrakt értékhalmaz és műveletek
- Ezek a műveletek is örökölhetők

Primitív műveletek

- „A típus megadása után felsorolt” alprogramok
  - Vagy valamelyik paraméter, vagy a visszatérési érték olyan típusú
- Tipikusan: egy csomagban definiálok egy (általában átlátszatlan) típust a műveleteivel
- Származtatott típusok: megöröklik a primitív műveleteket

Példa primitív műveletre

```
package Queues is
        type Queue( Capacity: Positive ) is limited private;
        procedure Hiext ( Q: in out Queue; E: in Element );
        procedure Lopop ( Q: in out Queue; E: out Element );
        …
private
        …
end Queues;
```

Példa öröklésre

Megörököltük
a
Hiext és Lopop
műveleteket

```
with Queues; use Queues;
package Dequeues is
        type Dequeue is new Queue;
        procedure Loext ( Q: in out Dequeue; E: in Element );
        procedure Hipop ( Q: in out Dequeue; E: out Element );
        …
end Dequeues;
```

Felüldefiniálás

- Egy típusra felüldefiniálhatjuk az előre definiált és a megörökölt műveleteket
  - (Előre definiált) operátorok
  - Primitív alprogramok
- Más implementációt rendelhetünk hozzájuk
- Ez különbözik a túlterheléstől

Túlterhelés és felüldefiniálás

```
package Racionálisok is
        type Racionális is private;
        function "+" ( P, Q: Racionális ) return Racionális;
        function "=" ( P, Q: Racionális ) return Boolean;
        …
private
        type Racionális is record
                                Számláló: Integer;
                                Nevező: Positive;
                        end record;
end Racionálisok;
```

túlterhelés

felüldefiniálás

Aggregátumok

- Összetett típusú (rekord, tömb) érték
- Jobbérték
  - értékadás jobb oldala, inicializálás
  - bemenő paraméter
- Pozícionális és névvel jelölt megfeleltetés
- Nagyon kényelmes
- Hasonló, de nem ilyen jó a C++-ban:

  int t[] = {3,5,6};

Rekordaggregátum

```
type Racionális is record … end record;
R1: Racionális := (3,2);
R2: Racionális := (Számláló => 3,  Nevező =>2);
R3: Racionális := (Nevező =>2,  Számláló => 3);
R4: Racionális := (3,  Nevező =>2);
function "*" (P, Q: Racionális) return Racionális is
begin
    return (P.Számláló * Q. Számláló,  P.Nevező * Q.Nevező);
end;
```

Tömbaggregátum

type T is array (1..6) of Float;

- Pozícionális megadás

  X: T := (1.0, 3.0, 1.0, 2.0, 2.0, 2.0);
- Névvel jelölt megadás

  X: T := (2 => 3.0, 1|3 => 1.0, 4..6 => 2.0);
- Maradék

  X: T := (2 => 3.0, 1|3 =>1.0, others => 2.0);
- Keverés

  X: T := (1.0, 3.0, 1.0, others => 2.0);

Korlátozás nélküli index esetén

type T is array (Integer range <>) of Float;

- Pozícionális megadás

  X: T := (1.0, 3.0, 1.0, 2.0, 2.0, 2.0);
- Névvel jelölt megadás

  X: T := (2 => 3.0, 1|3 => 1.0, 4..6 => 2.0);
- Helytelenek:

  X: T := (2 => 3.0, 1|3 =>1.0, others => 2.0);

  X: T := (1.0, 3.0, 1.0, others => 2.0);
- Helyesek:

  X: T(1..10) := (1.0, 3.0, 1.0, others => 2.0);

  X: T(1..10) := (2 => 3.0, 1|3 =>1.0, others => 2.0);

Többdimenziós esetben

M: Mátrix(1..2, 1..3):=

(1 = > (1.1,1.2,1.3), 2 = > (2.1,2.2,2.3));

D: Mátrix := (1 .. 5 = > (1 .. 8 = > 0.0));

A String típus

- Beépített típus

  type String is array (Positive range <>) of Character;
- Speciális szintaxis

  S1: constant String := "GIN";

  S2: constant String := ('G','I','N');
- Rugalmatlanabb, mint más nyelvekben
  - A tömbökre vonatkozó szabályok miatt
  - Mutatókkal, dinamikus memóriakezeléssel segíthetünk
  - Ada95: Bounded_String, Unbounded_String

Tipikus hibák String használatánál (1)

- Különböző méretű String-ek nem adhatók egymásnak értékül: Constraint_Error

```
S: String(1..256);
S := Integer'Image( Faktoriális(4) );
```

- Egy másik tanulság: lehetőleg ne égessünk bele a programba konstansokat.
  - Mi van, ha az Integer típus egy adott implementációban szélesebb?

Tipikus hibák String használatánál (2)

- Egy Get_Line-os beolvasás esetleg csak részben tölti fel a sztringet

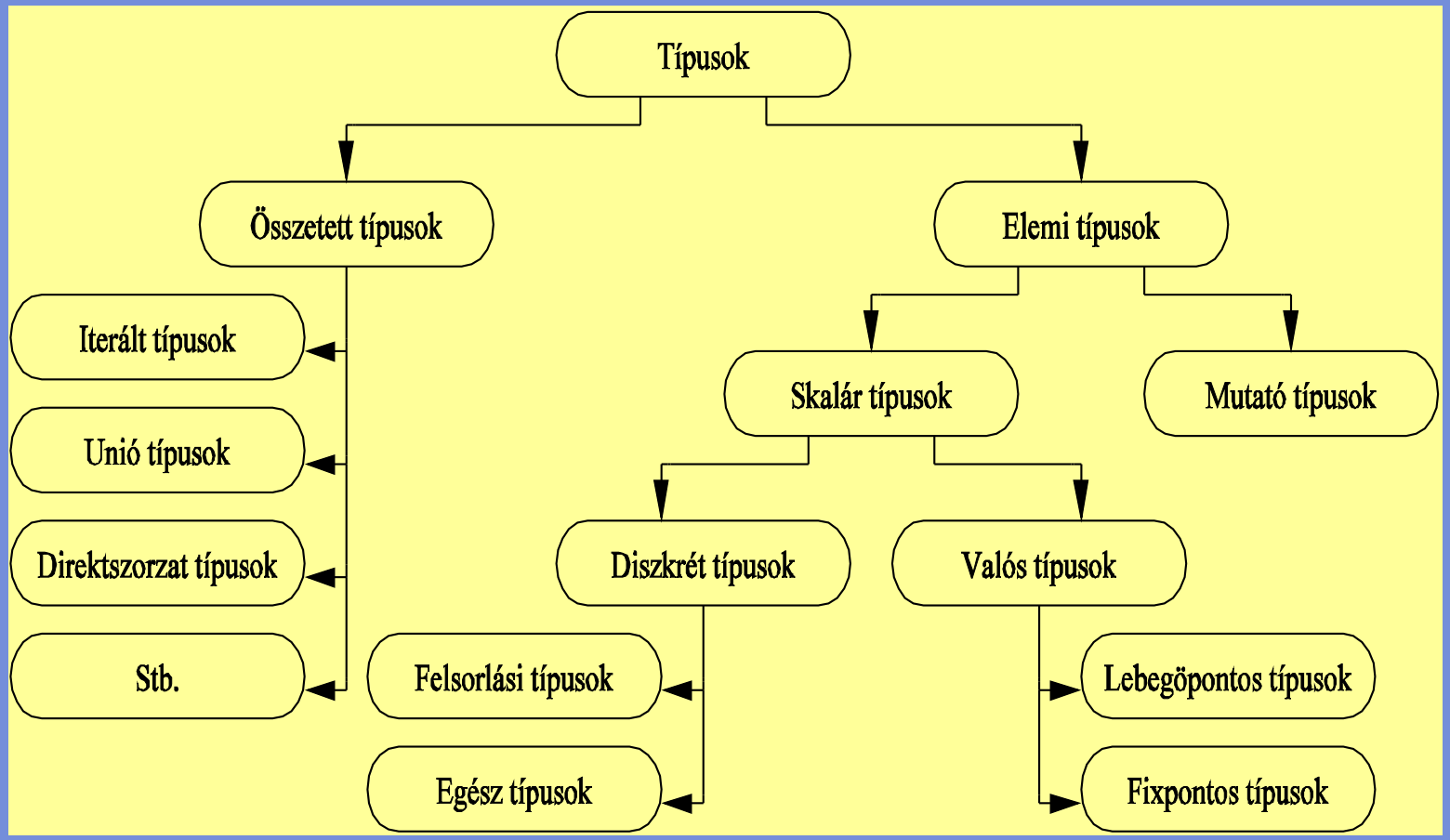
```
S: String( 1..Sor_Hossz );
H: Natural;
Get_Line(S,H);
… Integer'Value( S ) …
… Integer'Value( S(1..H) ) …
```

Szélesebb karakterkészlet

```
type Wide_String is array(Positive range <>)
                                of Wide_Character;
```

- A Wide_Character két bájton ábrázolja a karaktereket
- Ezek is beépített típusok

Típusosztályok rajzban



Elemi típusok

- skalár
  - diszkrét
    - felsorolási (Character, Boolean)
    - egész
      - előjeles (Integer)
      - **moduló típusok**
  - valós
    - **lebegőpontos** (Float)
    - **fixpontos**
      - **közönséges fixpontos**
      - **decimális fixpontos**
- mutató

Egész típusok (2) elemi, skalár, diszkrét

- Előjeles egész típusok, pl. *Integer*
- Moduló típusok

  type Mod10 is mod 10;
  type Bájt is mod 256;

  - A típusértékek a 0..9, illetve a 0..255
  - A számokon értelmezett szokásos műveletek (például a “+”) a maradékosztályok szerint, azaz modulo számolnak.

Valós típusok (1) elemi, skalár

- Lebegőpontos számtípusok
  - egy rögzített hosszúságú *mantissza* és egy előjeles egész *exponens*

    type Real is digits 8;

    R: Real := 2.12345678
  - a mantissza ábrázolásához szükséges decimális jegyek száma
  - predefinit típus: Float - implementációfüggő
  - a többi ebből, származtatással

    type Real is new Float digits 8;
- Fixpontos számtípusok

Valós típusok (2) elemi, skalár

- Lebegőpontos számtípusok
- Fixpontos számtípusok
  - rögzített számú számjegy és egy képzeletbeli tizedespont

    type Fix is delta 0.01 range -1.0 .. 1.0;
  - tizedes fixpontos típus:

    type Fix is delta 0.01 digits 15;

    az értékek a következő intervallumban:

    –(10**digits–1)*delta .. +(10**digits–1)*delta.

A skalár típusosztály attribútumai

S'First, S'Last, S'Range, S'Image, S'Value

S'Base	az S bázistípusa (a megszorítás nélküli altípus)
S'Min	function S'Min(A,B: S'Base) return S'Base
S'Max	a két érték maximuma
S'Succ	function S'Succ(A: S'Base) return S'Base rákövetkező elem (Constraint_Error lehet)
S'Pred	…
S'Width	maximuma az S'Image által visszaadott stringek hosszának

S'Wide_Image, S'Wide_Width, S'Wide_Value

A diszkrét típusosztály

- Felsorolási és egész (előjeles és moduló) típusok
- Ezen típusoknál a típusértékek a típuson belül pozíciószámmal azonosíthatók

S'Pos(A) function S'Pos(A: S'Base) return Integer
egy egész szám, a pozíció

S'Val(n) az adott pozíciónak megfelelő típusérték

- A pozíciószám követi a skalár rendezést

A felsorolási típusok osztálya

- A skalár rendezést itt a típusértékek felsorolási sorrendje adja
- A diszkrét pozíciószám szintén a felsorolást követi
  - nullától egyesével

Az egész típusok osztálya

- Predefinit operátorok

  +A -A A+B A-B A*B A/B

  A rem B A mod B abs A A**B

  - Az egész osztás csonkít (nulla felé...)
  - Hatványozásnál B nemnegatív
- Moduló típusoknál:

  S'Modulus a típus modulusa

A valós típusok osztálya

+X -X X+Y X-Y X*Y X/Y X**Y

- ***hatványozás:*** **Y *egész***

- Attribútumok
  - Lebegőpontos: *'Digits*
  - Fixpontos: *'Small, 'Delta, 'Fore, 'Aft, 'Scale, 'Round*
    - Decimális fixpontos: *'Digits*

Appendix

Név szerinti paraméterátadás

- Call-by-name
- Archaikus: Algol 60, Simula 67
- Az aktuális paraméter kifejezést újra és újra kiértékeljük, ahányszor hivatkozás történik a formálisra
  - A törzs kontextusában értékeljük ki
  - A formális paraméter különböző előfordulásai mást és mást jelenthetnek az alprogramon belül

Jensen's device

```
real procedure sum ( expr, I, low,
    high );
    value low, high;
    real expr;
    integer I, low, high;
begin
    real rtn;
    rtn := 0;
    for I := low step 1 until high do
            rtn := rtn + expr;
    sum := rtn;
end sum
```

$$y = \sum_{i=1}^{10} 3x^2 - 5x + 2$$

```
y := sum (3*x*x – 5*x + 2, x, 1, 10)
```

További paraméterátadási módok

- Megosztás szerinti
  - Call-by-sharing
  - Objektumelvű nyelvek (CLU, Eiffel, Java…)
  - Szintaxisban: érték szerinti, valójában olyan, mint a cím szerinti (alias)
- Igény szerinti
  - Call-by-need
  - Lusta kiértékelésű funkcionális nyelvek (Miranda, Haskell, Clean)
  - Az aktuálist nem híváskor értékeli ki, hanem akkor, amikor szüksége van rá a számításokhoz

Tömbaggregátum

```
type Tömb is array (Napok range <>) of Natural;
T1: Tömb(Kedd..Péntek) := (1,9,7,0);
T2: Tömb := (1,9,7,0);              -- Hétfőtől indexelve
T3: Tömb(Kedd..Szerda) := (Kedd => 5, Szerda => 2);
T4: Tömb(Kedd..Szerda) := (Szerda => 2, Kedd => 5);
T5: Tömb := (Szerda => 2, Kedd => 5);
T6: Tömb := ( Kedd | Vasárnap => 5, Szerda..Szombat => 1);
T7: Tömb(Hétfő..Szombat):= (Hétfő..Szerda=>5, others=>1);
T8: Tömb(Hétfő..Szombat) := ( 2, 4, 5, others => 1);
```

Rekord aggregátum

- *pozíció szerinti forma*:

  D := (1848, Március, 15);

- *név szerinti forma:*

  D := (Nap => 15, Hó => Március, Év => 1848);

- keverhetjük is:

  D := (1848, Nap => 15, Hó => Március);

- Ha típusa nem egyértelmű: minősítés

  Ma: Date := Date (Dátum'(1848,Március,15));