SOL (**S**tructured **O**dd **L**anguage)

- Imperative paradigm
- Declaration of *types*, *variables*, *constants*, *functions*
- Program structured in nested functions
- Static scope
- In-mode parameter passing
- Function body = block of statements
- Recursion
- Data types: atomic, structured (*struct*, *vector*)
- Instance constructors (**struct**, **vector**)

Atomic Types

- char

```
c : char;
c = 'a';
```

- int

```
i : int;
i = 25;
```

- real

```
r : real;
r = 64.15
```

- string

```
s : string;
s = "alpha";
```

- bool

```
ok : bool;
ok = true;
```

Struct Type

- `struct`

```
person: struct(name, surname: string; birth: int;);
person = struct("John", "Smith", 1987);
... person.name == "John" ...
```

- Orthogonality:

```
book: struct(author: struct(name, surname: string;);
             title: string;
             edition: struct(editor: string; year: int;););

book = struct(struct("Carl Gustav", "Jung"),
              "Man and his symbols",
              struct("ETA", 1992));

... book.edition.year == 1992 ...
```

Vector Type

- `vector`

```
numbers: vector [10] of int;
numbers = vector(5, 29, 13, 25, 67, 123, 3, 45, 78, 12);
```

- Range of indexing: [0 .. (*dimension* - 1)]

```
... numbers[3] == 25 ...
```

- Orthogonality:

```
matrix: vector [3] of vector [5] of int;
matrix = vector(vector(1,2,3,4,5),
                vector(2,4,6,8,10),
                vector(3,6,9,12,15));
... matrix[1][2] == 6 ...
```

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15

```
people: vector [4] of struct(name, surname: string; birth: int;);
people = vector(struct("John", "Smith", 1983),
                struct("Rosy", "White", 1960),
                struct("Louis", "Green", 1998),
                struct("Ann", "Black", 2001));
... people[2].surname == "Green" ...
```

Generality of Instance Constructors

- Possible to apply instance constructors (**`struct`**, **`vector`**) to expressions

- Constraints: involved expressions shall have
  - correct type (*static constraint*)
  - defined value (*dynamic constraint*)

```
v: vector [4] of int;
i,j,k: int;

v = vector(3, i, j+k, (i-j)*k);
v = vector(v[4], v[3], v[2], v[1]);
```

```
t: struct(a: string; b: vector[4] of int; c: real; d: int;);
s: string,
x: real;
i,j,k: int;
v: vector[4] of int;

v = vector(i,j,i+j-k,10);
t = struct(s, v, f(x), v[i-j]+k);
```

Declarations

- **type** section:

```
type  T1: int;
      T2: string;
      T3, T4: T2;
      T5: vector [10] of T3;
```

- **var** section:

```
var   i, j: int;
      z: T1;
      s, t: T5;
      a: vector [100] of int;
```

- **const** section (constant values, even if constructed by expressions):

```
const
  MAX: int = 100;
  name: T2 = "alpha";
  VECT: vector [5] of real = vector(2.0, 3.12, 4.67, 1.1, 23.0);
  MAT: vector [2] of vector [5] of real = vector(VECT, vector(x, y, z, 10.0, x+y+z));
```

Program Structure

```
func prog(a: int; b: string;): int
  type T1: …, T2: …, Tn: …;
  var v1: …, v2: …, vm: …;
  const c1: … = …, c2: … = …, cp: … = …;

  func f1(…): T1
    type …;
    var …;
    const …;

    func f2(…): T2
      ...
    begin f2
      ...
    end f2

  begin f1
    ...
  end f1

  ...

begin prog
  ...
end prog
```

program parameters:
input from keyboard
output to monitor

- Not required function declaration before call

Arithmetic Expressions

- +, –, ∗, / : overloaded operators applicable to { **int** / **real** }
- Mixed expressions not allowed (no coercion)
- Cast operators { **toint** : **real → int** / **toreal** : **int → real** }

```
i,j: int;
x, y: real;
…
x = toreal(i+j)*(r-toreal(i));
j = toint(x+y–1.25);
```

Relational Operators

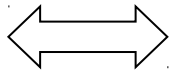
- **`==, !=`** : applicable to all types
- **`>, >=, <, <=`** : applicable to `int`, `real`, `string`
- **`in`**: membership, where second operand of `vector` type
- Non-atomic types $\rightarrow$ compatibility by structure

Logical Expressions

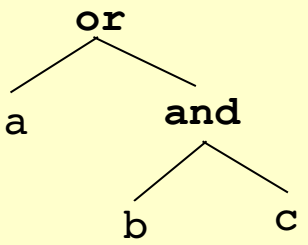
- **`and`**, **`or`**, **`not`**: applicable to **`bool`**

- Short-circuit evaluation

```
a, b, c, d: bool;
...
d = a or (b and c);
```



```
if a then
  d = true
else
  if not b then
    d = false
  else
    d = c
  endif
endif
```

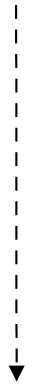


- Integration with relational operations

```
i, j: int;
lastname: string;
person: struct(name, surname: string;);
a, b: bool;
…
b = (i==j+2 or a) and (struct("maria", lastname) == person);
```

Precedence, Associativity, Evaluation Order

Operator	*Type*	*Associativity*
`and`, **`or`**	binary	left
`==`, **`!=`**, **`>`**, **`>=`**, **`<`**, **`<=`**, **`in`**	binary	nonassoc
`+`, **`-`**	binary	left
`*`, **`/`**	binary	left
`-`, **`not`**	unary	right



increasing precedence

- Evaluation order of operands: from left to right

Conditional Expression

- **if** *expr* **then** *expr* { **elsif** *expr* **then** *expr* } **else** *expr* **endif**

```
a, b, c: int;

a = if b>c then  b+c elsif b==c+1 then b-c else a+1 endif;
```

Conditional Statement

- **if** *expr* **then** *stat-list* { **elsif** *expr* **then** *stat-list* } [**else** *stat-list*] **endif**

```
a, b, c: int;
t, r: struct(x: int; y: string;);
…
if a==b then
    t = r;
elsif a>b then
    t = struct(2, "alpha");
    a = b + c;
else
  a = b-c;
endif;
```

While Loop

- **`while`** *expr* **`do`** *stat-list* **`endwhile`**

```
a, b, res: int;
…
res = 0;
while a >= b do
    res = res + 1;
    a = a – b;
endwhile;
```

For Loop

- **`for id =`** *expr* **`to`** *expr* **`do`** *stat-list* **`endfor`**

```
v: vector [100] of int;
i: int;

for i=1 to 100 do
   k = f(v[i]);
   p(k, i);
endfor;
```

- Counting variable (type **`int`**) not assignable within loop body.

- *expr*: of `int` type, evaluated just once (before iteration).

Foreach Loop

- **`foreach id in`** *expr* **`do`** *stat-list* **`endforeach`**

```
v: vector [100] of int;
m, n: int;

foreach n in v do
   m = compute(n);
endforeach;
```

- *expr*: of type `vector`, evaluated just once (before iteration).

Return

- **`return`** *expr*

```
func sum(v: vector [100] of int): int
var n, res: int;
begin sum
  res = 0;
  foreach n in v do
    res = res + n;
  endforeach;
  return res;
end sum
```

```
func fact(n: int): int
begin fact
  if n <= 0 then
    return 1;
  else
    return n * fact(n-1);
  endif;
end fact
```

Input

- **`read` id**
- **`read [`** *filename* **`]` id**

```
v: vector [100] of int;
...
read v;
...
read ["v.dat"] v;
...
```

- **`rd`** *domain*
- **`rd [`** *filename* **`]`** *domain*

```
type
  Vect: vector [100] of int;
var
  name: string;
  v1, v2: Vect;
...
v1 = rd Vect;
v2 = reverse(rd [name] Vect);
```

Output

- **write** *expr*
- **write [** *filename* **]** *expr*

```
i, j: int;
name: string;
v: vector [100] of int;
…
write f(v);
read name;
write [name] reverse(v);
```

- **wr** *expr*
- **wr [** *filename* **]** *expr*

```
i: int;
name: string;
v: vector [100] of int;
v1, v2: vector [20] of int;
...
v1 = wr f(v);
...
v2 = f(wr [name] v);
```