

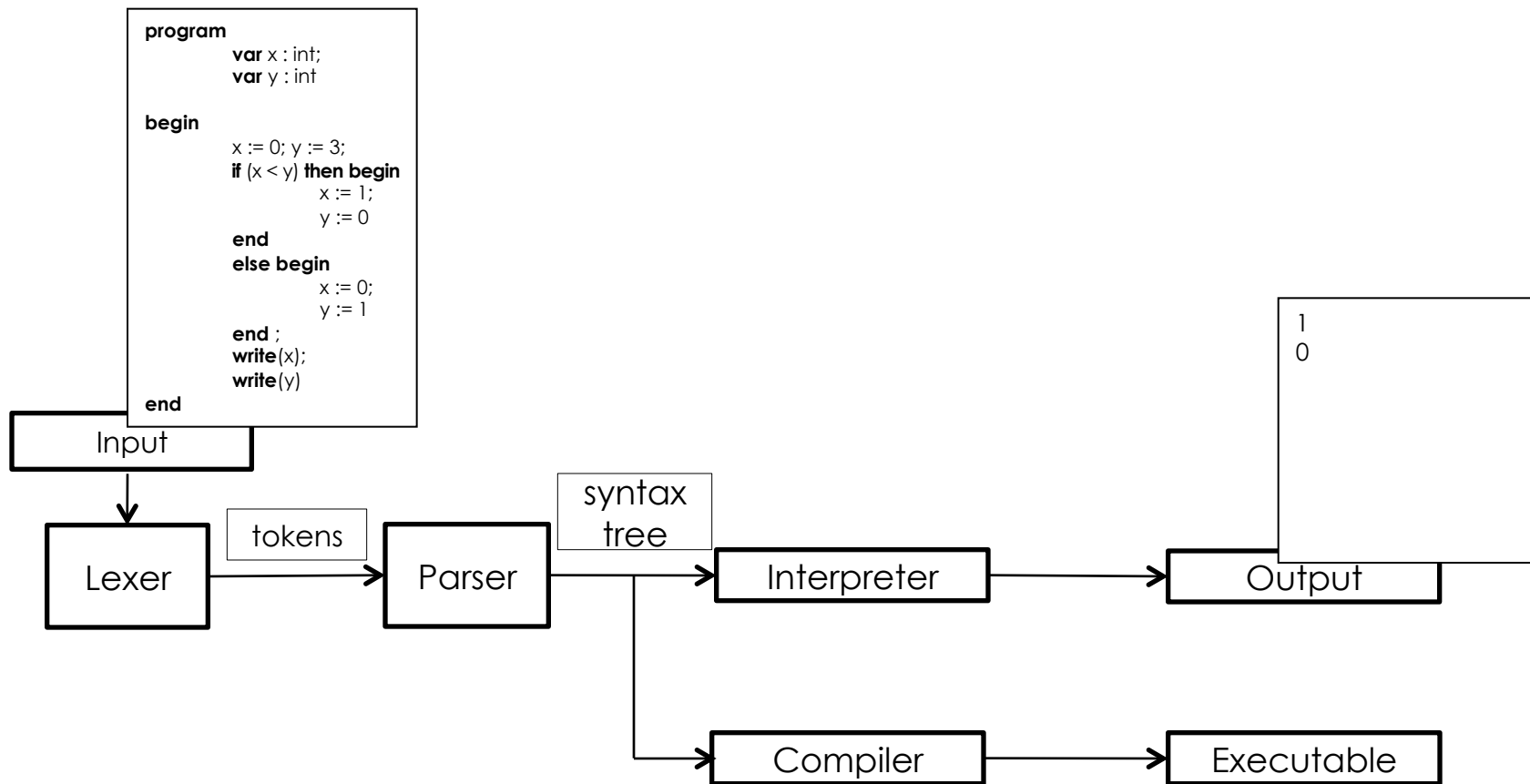
# Revisiting Interpreter

Lecture 15

Formal Languages and Compilers 2011

Nataliia Bielova

# Interpreter $\neq$ Compiler



# Front-end analysis (the same for interpreter and compiler)

3

Recognition of tokens (lexer)



Checking the syntax (parser)



Building the syntax tree (parser)

# 1 faze of interpreter

Syntax tree



Execution of syntax tree

# How interpreter is made

- parser.mly: definition of tokens
- lexer.mll: regular expressions and creation of tokens
- syntaxtree.ml: declarations of types for the syntax tree
- parser.mly: language grammar and creation of the syntax tree
- mail.ml: starts lexer, parser, executes syntax tree
- interpreter\_base.ml: functions for the execution of the syntax tree

# Definition of the memory and environment

6

- Formal definition:

$Store : Loc \rightarrow Val$

type store = loc -> value

$Env : Id \rightarrow (Loc \cup Val)$

type env = ide -> env\_entry

- Updating the memory:

$$\text{updatemem}(s, l, v)(x) = \begin{cases} v & \text{if } x = l \\ s(x) & \text{if } x \neq l \end{cases}$$

let updatemem((s:store), addr, (v:value)): store = function

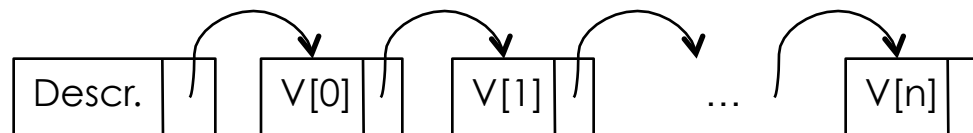
x -> if (x = addr) then v else s(x)

# Implementation of vectors



☺  $\Lambda \|V[k]\| = B + O(k)$

☹ Ins. & Del.



☹  $\Lambda \|V[k]\| = \text{scanning the whole list}$

☺ Ins. & Del.

# Vectors

- `var V:array [LB .. UB] of type`
- Address of the k-th element:

$$\Lambda\|V[k]\| = \alpha + (k - LB) = (\alpha - LB) + k = VO + k$$

$$VO = \alpha - LB = \Lambda\|V[0]\|$$

Descriptor:

VO
LB
UB

Representation in the memory:

V[0]
...
V[LB]
V[LB+1]
...
V[UB]

VO

$\alpha$

```
var V : array[3..5] of int;
```

$$\Lambda\|V[4]\| = \alpha + (4 - 3) = VO + 4$$

$$VO = \alpha - 3 = \Lambda\|V[0]\|$$



# Bidimensional matrices

`var V : array[LB1..UB1, LB2..UB2] of type`

- Dimension of an element:  $M_2$
- Dimension of a row:  $M_1 = (UB_2 - LB_2 + 1) \times M_2$
- Virtual Origin:  $VO = \alpha - LB_1 \times M_1 - LB_2 \times M_2$

$$\|V[i, j]\| = VO + i \times M_1 + j \times M_2$$

# Virtual Origin (V.O.)

```
var v : array[5..8][2..6] of int;
```

$$M_2 = \text{sizeof}(\text{int}) = 1$$

$$\begin{aligned} M_1 &= (\text{UB}_2 - \text{LB}_2 + 1) \times M_2 = \\ &= (6 - 2 + 1) \times M_2 = 5 \end{aligned}$$

$$\begin{aligned} \text{VO} &= \alpha - \text{LB}_1 \times M_1 - \text{LB}_2 \times M_2 = \\ &= \alpha - 5 \times M_1 - 2 \times M_2 = \alpha - 27 \end{aligned}$$

$$\begin{aligned} \Lambda\|v[7, 4]\| &= \text{VO} + 7 \times M_1 + 4 \times M_2 = \\ &= \text{VO} + 35 + 4 \end{aligned}$$

VO

	$\alpha$	2	3	4	5	6
	5					
	6					
	7					
	8					

# Virtual Origin (V.O.)

```
var v : array[5..8][2..6] of int;
```

$$M_2 = \text{sizeof}(\text{int}) = 1$$

$$\begin{aligned} M_1 &= (\text{UB}_2 - \text{LB}_2 + 1) \times M_2 = \\ &= (6 - 2 + 1) \times M_2 = 5 \end{aligned}$$

$$\begin{aligned} \text{VO} &= \alpha - \text{LB}_1 \times M_1 - \text{LB}_2 \times M_2 = \\ &= \alpha - 5 \times M_1 - 2 \times M_2 = \alpha - 27 \end{aligned}$$

$$\begin{aligned} \Lambda\|v[7, 4]\| &= \text{VO} + 7 \times M_1 + 4 \times M_2 = \\ &= \text{VO} + 35 + 4 \end{aligned}$$

VO

$\alpha$	0	1	2	3	4	
5						
6						
7						
8						

# Virtual Origin (V.O.)

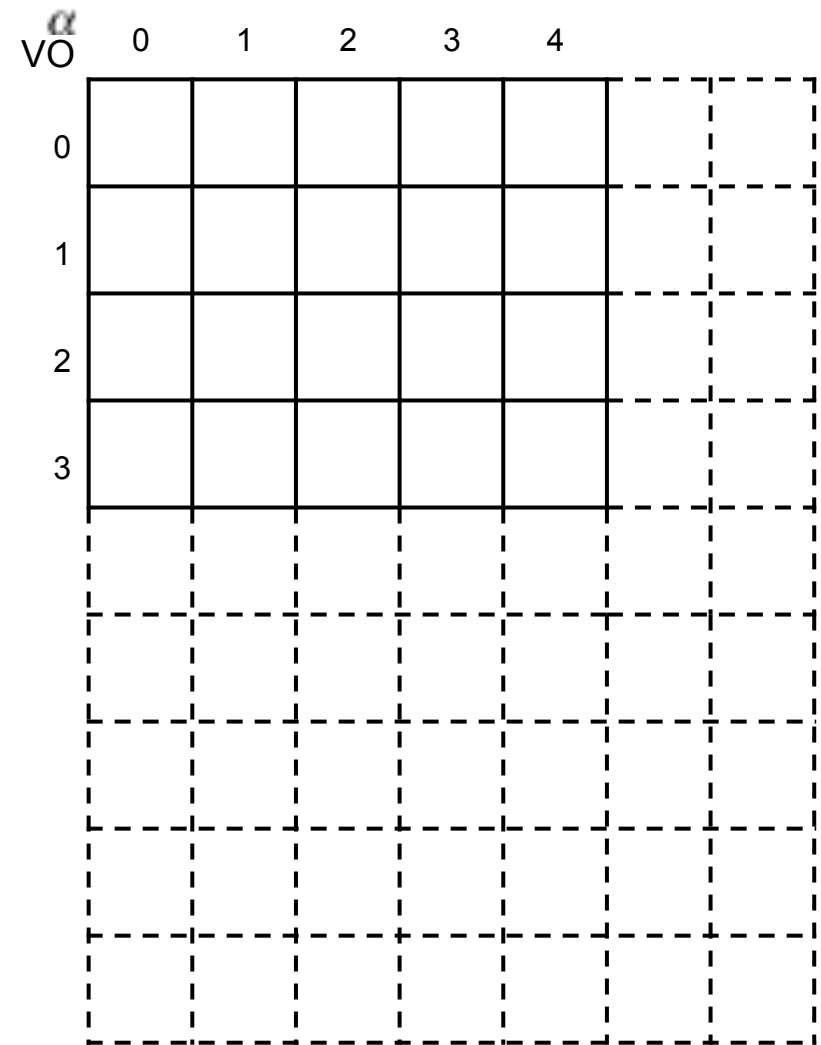
```
var v : array[5..8][2..6] of int;
```

$$M_2 = \text{sizeof}(\text{int}) = 1$$

$$\begin{aligned} M_1 &= (\text{UB}_2 - \text{LB}_2 + 1) \times M_2 = \\ &= (6 - 2 + 1) \times M_2 = 5 \end{aligned}$$

$$\begin{aligned} \text{VO} &= \alpha - \text{LB}_1 \times M_1 - \text{LB}_2 \times M_2 = \\ &= \alpha - 5 \times M_1 - 2 \times M_2 = \alpha - 27 \end{aligned}$$

$$\begin{aligned} \Lambda\|v[7, 4]\| &= \text{VO} + 7 \times M_1 + 4 \times M_2 = \\ &= \text{VO} + 35 + 4 \end{aligned}$$



# Multidimensional matrices

`var V : array[LB1..UB1, ..., LBn..UBn] of type`

- Multipliers:

$$M_n = M$$

$$M_i = (UB_{i+1} - LB_{i+1} + 1) \times M_{i+1} \quad i \in [1, n - 1]$$

$$VO = \alpha - \sum_{i=1}^n LB_i \times M_i$$

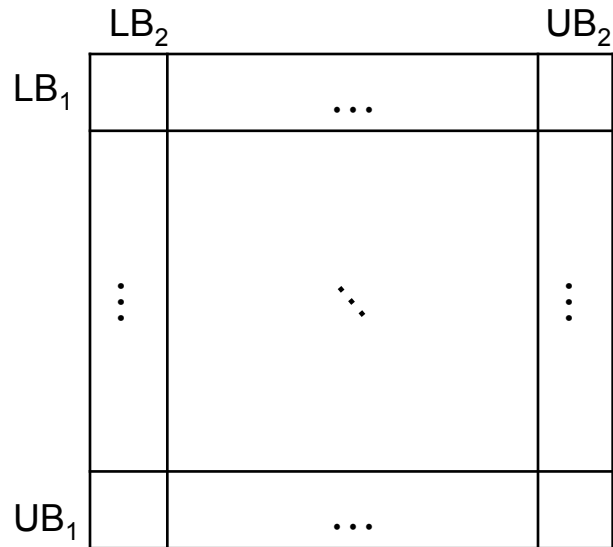
$$\Lambda[V[k_1, \dots, k_n]] = VO + \sum_{i=1}^n k_i \times M_i$$

`array[LB1..UB1, LBn..UBn] of type`

`=`

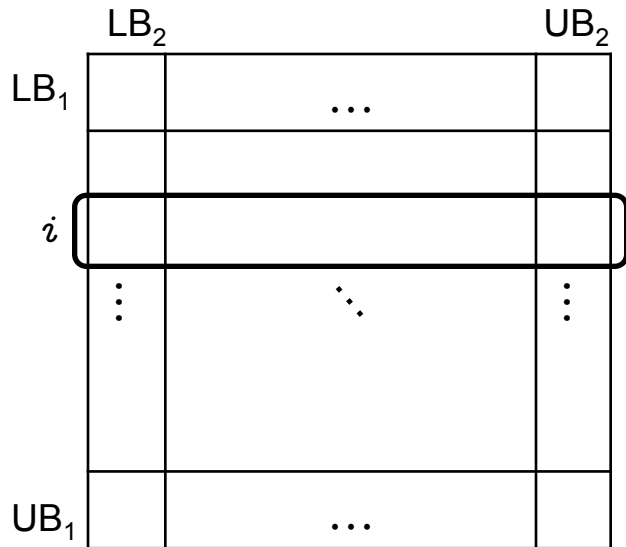
`array[LB1..UB1] of (array[LB2..UB2, LBn..UBn] of type)`

# Slices of array



```
var v:array[LB1 ..UB1 ,LB2 ..UB2] of type;  
var s: slice[i,*] of v;
```

# Slices of array



$$M = M_2$$

$$VO_I = VO_V + i \times (UB_2 - LB_2 + 1) \times M_2 =$$

$$VO_V + i \times M_1$$

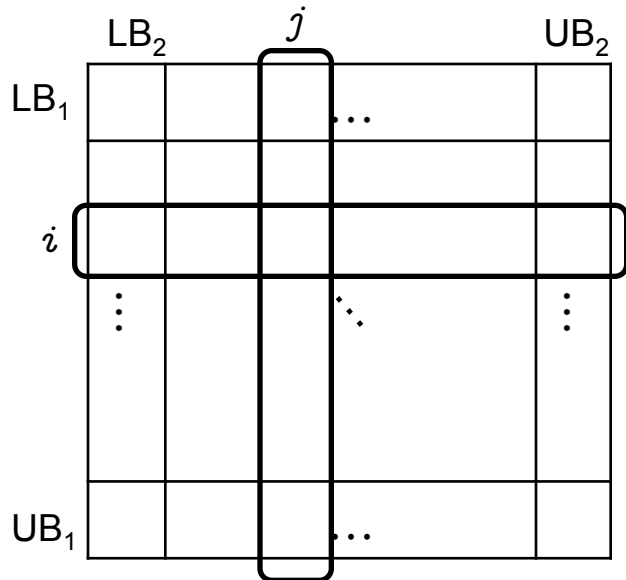
$$LB = LB_2$$

$$UB = UB_2$$

$$\Lambda \|I[k]\| = VO_I + k \times M$$

$$I = V[i][*] = \{V[i][LB_2], V[i][LB_2 + 1], \dots, V[i][UB_2]\}$$

# Slices of array



$$M = (UB_2 - LB_2 + 1) \times M_2 = M_1$$

$$VO_j = VO_v + j \times M_2$$

$$LB = LB_1$$

$$UB = UB_1$$

$$A[J[k]] = VO_j + k \times M$$

$$I = V[i][*] = \{V[i][LB_2], V[i][LB_2 + 1], \dots, V[i][UB_2]\}$$

$$J = V[*][j] = \{V[LB_1][j], V[LB_1 + 1][j], \dots, V[UB_1][j]\}$$



# Types of passing the parameters

Notation `call P(x  $\leftarrow_{\alpha}$  e)` means that

- P is declared as `proc P(x) ...`
- P is invoked as `call P(e)`
- $\alpha$  is type of passing the parameters

Value	<code>call P(x <math>\leftarrow_{\text{Val}}</math> e)</code>
Value-result	<code>call P(x <math>\leftarrow_{\text{Val-res}}</math> y)</code>
Result	<code>call P(x <math>\leftarrow_{\text{Res}}</math> y)</code>
Reference	<code>call P(x <math>\leftarrow_{\text{Ref}}</math> y)</code>
Constant	<code>call P(x <math>\leftarrow_{\text{Const}}</math> e)</code>
Name	<code>call P(x <math>\leftarrow_{\text{Name}}</math> e)</code>

Note: x, y are variables, e is an arithmetical expression

# Passing by name

`call P(x  $\leftarrow_{\text{Name}}$  e)`

- create a new couple  $\langle e, \mathfrak{r} \rangle$ , where  $\mathfrak{r}$  is an environment of the caller
- every time when  $x$  should be evaluated,  $e$  is getting evaluated instead in the environment  $\mathfrak{r}$  and put instead of  $x$ .
- $x$  cannot be assigned values in  $P$