

Interpreter for “crème CAraMeL”

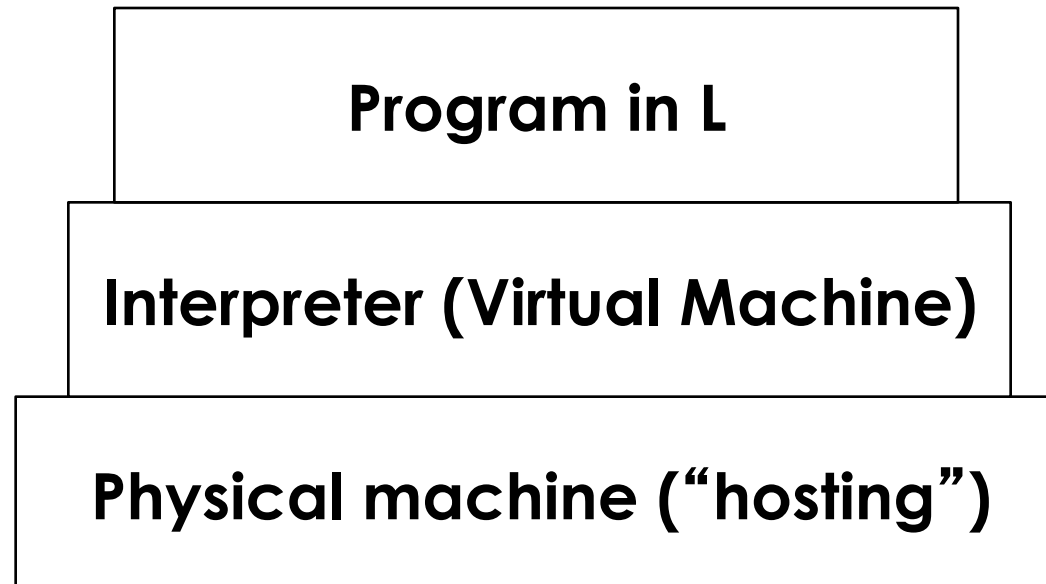
Lecture 5

Formal Languages and Compilers 2011

Nataliia Bielova

Definition

- Interpreter for a language L:



crème CARAMEL

- Basic types: int and float
- Flow control: if then else, while do, for
- Arithmetic operators: +, -, *, /
- Assignment: ::=
- Relational operators: =, <, <=
- Boolean operators: &, |, !
- Utility: write(val)

Objective

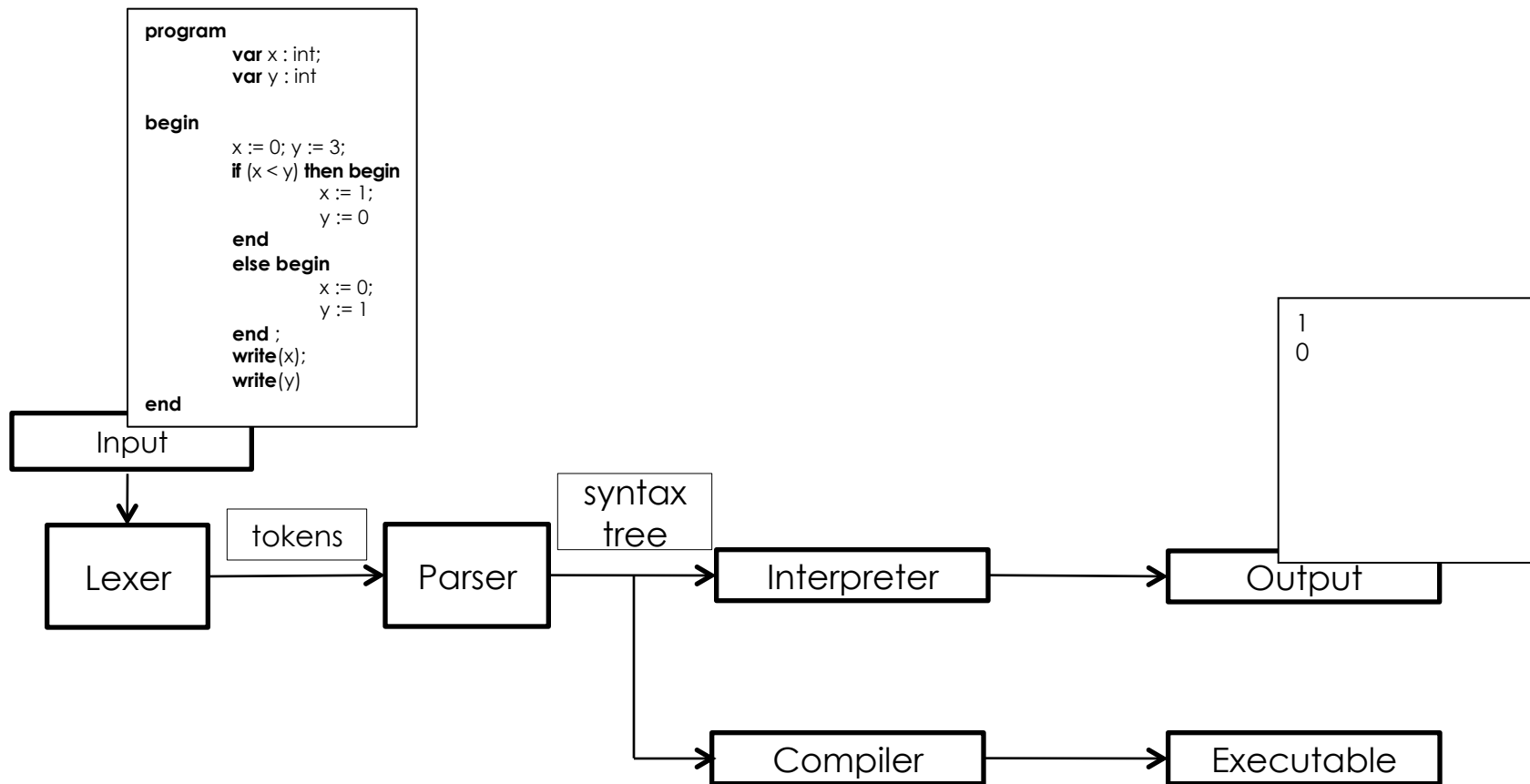
- Construct an interpreter for the language crème CAraMeL

```
program  
  var x : int;  
  var y : int  
  
begin  
  x := 0; y := 3;  
  if (x < y) then begin  
    x := 1;  
    y := 0  
  end  
  else begin  
    x := 0;  
    y := 1  
  end ;  
  write(x);  
  write(y)  
end
```

Interpreter
⇒

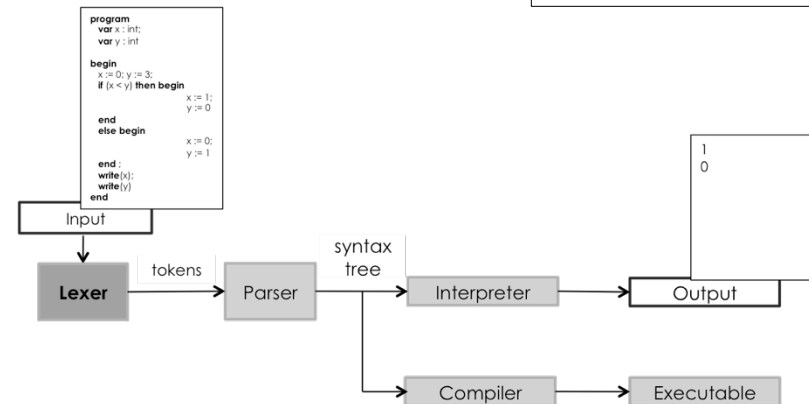
```
1  
0
```

Interpreter or compiler?



Elements of interpreter

- Lexer: **in**: input, **out**: token



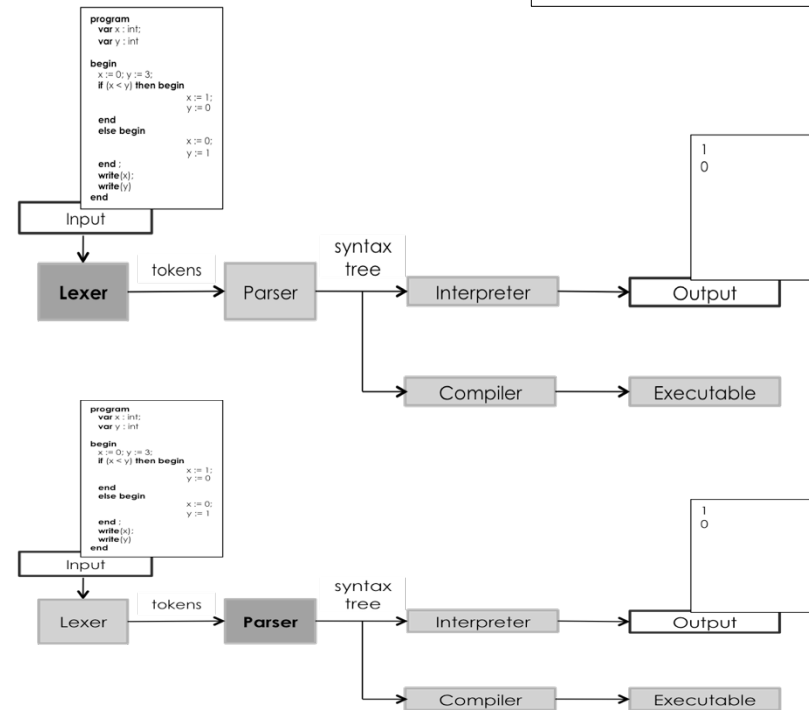
Parser: **in**: token, **out**: abstract syntax tree (a.s.t.)

Interpreter itself (??): **in**: a.s.t, **out**: output

Elements of interpreter

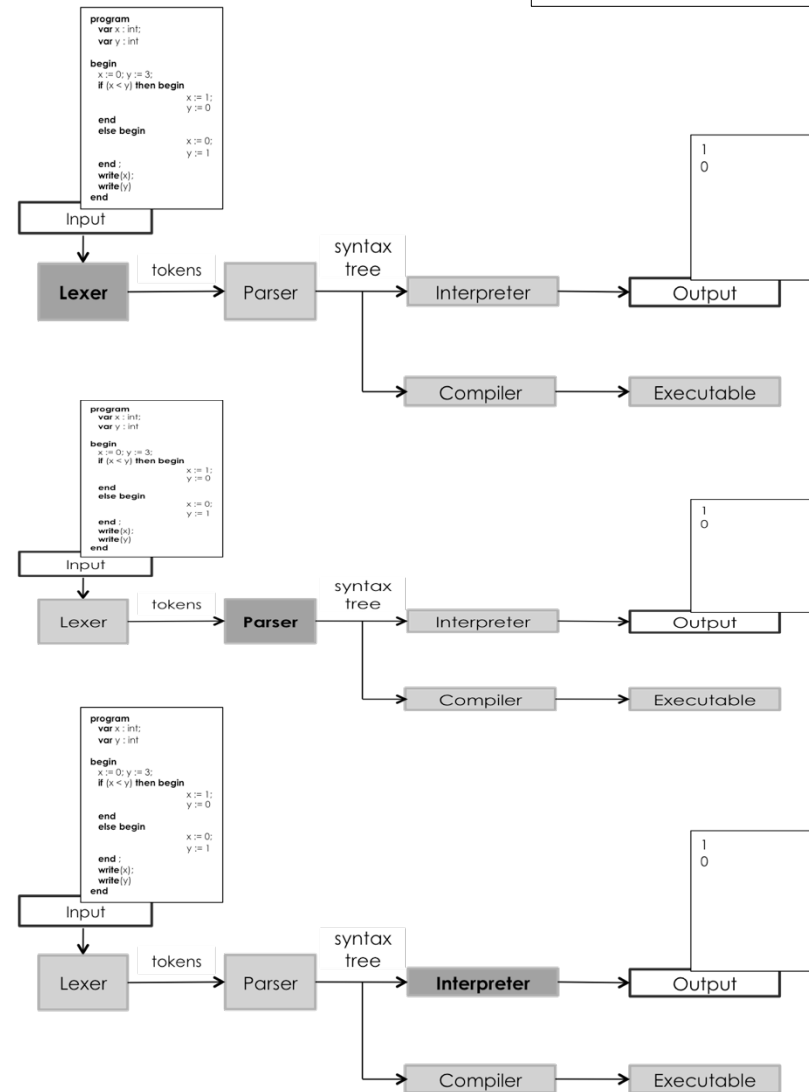
- Lexer: **in:** input, **out:** token
- Parser: **in:** token, **out:** abstract syntax tree (a.s.t.)

Interpreter itself (??): **in:** a.s.t, **out:** output



Elements of interpreter

- Lexer: **in:** input, **out:** token
- Parser: **in:** token, **out:** abstract syntax tree (a.s.t.)
- Interpreter itself: **in:** a.s.t, **out:** output



Base of the interpreter

http://disi.unitn.it/~bielova/flc/exercises/05-Interpreter_base.zip

- Definition of the lexer: lexer.mll
- Definition of the parser: parser.mly
- Definition for a.s.t: syntaxtree.ml
- Definition of the interpreter: interpreter_base.ml
- Main program: main.ml

Compilation:

`make eval # compiles everything (win: make.bat)`

`make clean # “cleans” from the compiled files (win: clean.bat)`

`./interpreter_base # starts the interpreter (input from console)`

`./interpreter_base < input/test_1.cre # interprets the input from test 1`

Base of the interpreter

http://disi.unitn.it/~bielova/flc/exercises/05-Interpreter_base.zip

- Definition of the lexer: lexer.mll
- Definition of the parser: parser.mly
- Definition for a.s.t: syntaxtree.ml
- Definition of the interpreter: interpreter_base.ml
- Main program: main.ml

Compilation:

```
./make.bat # compiles everything
```

```
./clean.bat # "cleans" from the compiled files
```

```
./interpreter_base # starts the interpreter (input from console)
```

```
./interpreter_base < input/test_1.cre # interprets the input from test 1
```

How interpreter is made

- parser.mly: definition of tokens

How interpreter is made

- parser.mly: definition of tokens
- lexer.mll: regular expressions and creation of tokens

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

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

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

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

Semantic analysis

evaluation of expressions and declarations,
execution of commands

Definition of the memory and environment

- 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)

Arithmetic and boolean expressions: evaluation

$$E : Aexpr \times Env \times Store \rightarrow Val$$

$$E \parallel \text{Sum}(n_1, n_2) \parallel_{rs} = E \parallel n_1 \parallel_{rs} + E \parallel n_2 \parallel_{rs}$$

$$E \parallel i \parallel_{rs} = \begin{cases} s(r(i)) & \text{if } r(i) \in Loc \\ r(i) & \text{if } r(i) \in Val \end{cases}$$

$$B : Bexp \times Env \times Store \rightarrow \{\text{true}, \text{false}\}$$

$$B \parallel \text{Or}(b_1, b_2) \parallel_{rs} = \begin{cases} \text{true} & \text{if } B \parallel b_1 \parallel_{rs} \text{ is true} \\ B \parallel b_2 \parallel_{rs} & \text{otherwise} \end{cases}$$

Declaration: evaluation

$D : Decl \times Env \times Store \rightarrow Env \times Store$

$D \parallel \text{const } v = n \parallel_{rs} = r's$

where:

$$r'(y) = \begin{cases} r(y) & \text{if } y \neq v \\ n & \text{if } y = v \end{cases}$$

$D \parallel \text{var } v := n \parallel_{rs} = r's'$

where:

$$r'(y) = \begin{cases} r(y) & \text{if } y \neq v \\ 1 & \text{if } y = v \end{cases}$$

$$s'(x) = \begin{cases} s(x) & \text{if } x \neq 1 \\ n & \text{if } x = 1 \end{cases}$$

$1 = \text{newmem}(s)$ location that
is not used in s

Commands: execution

$C : Com \times Env \times Store \rightarrow Store$

$$C \parallel X := e \parallel_{rs} = s'$$

where :

$$l = A \parallel X \parallel_{rs}$$

$$v = E \parallel e \parallel_{rs}$$

$$s' = \text{updatemem}(s, l, v)$$

$$C \parallel \text{if } b \text{ then } c_1 \text{ else } c_2 \parallel_{rs} = s'$$

where :

$$s' = \begin{cases} C \parallel c_1 \parallel_{rs} & \text{if } B \parallel b \parallel_{rs} = \text{true} \\ C \parallel c_2 \parallel_{rs} & \text{otherwise} \end{cases}$$

$$C \parallel \text{while } b \text{ do } c \parallel_{rs} = \begin{cases} s & \text{if } B \parallel b \parallel_{rs} = \text{false} \\ C \parallel \text{while } b \text{ do } c \parallel_{rs'} & \text{otherwise} \end{cases}$$

$$\text{where } s' = C \parallel c \parallel_{rs}$$

Example: repeat - until

```
repeat
  cmd
until bexp
```

$$C \parallel \text{repeat cmd until bexp} \parallel_{rs} = s'$$

where :

$$s' = \begin{cases} s'' & \text{if } E \parallel \text{bexp} \parallel_{rs'} = \text{true} \\ C \parallel \text{repeat cmd until bexp} \parallel_{rs'} & \text{otherwise} \end{cases}$$

$$s'' = C \parallel \text{cmd} \parallel_{rs}$$

Example: repeat - until

```
repeat
  cmd
until bexp
```

- parser.mly: token REPEAT and UNTIL
- lexer.mll: strings "repeat" and "until"
- syntaxtree.ml: constructor Repeat of cmd * bexp for type cmd
- parser.mly: production REPEAT cmd UNTIL bexp { ... } for non-terminal symbol cmd
- main.ml: nothing :)
- interpreter_base.ml: execution of the command repeat - until

Programming in crème CAraMeL!

- Function for the Fibonacci number:

$$fib(n) = \begin{cases} n & \text{if } n < 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Factorial of the number:

$$n! = \begin{cases} 0 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$