Interpreter for "crème CAraMeL"

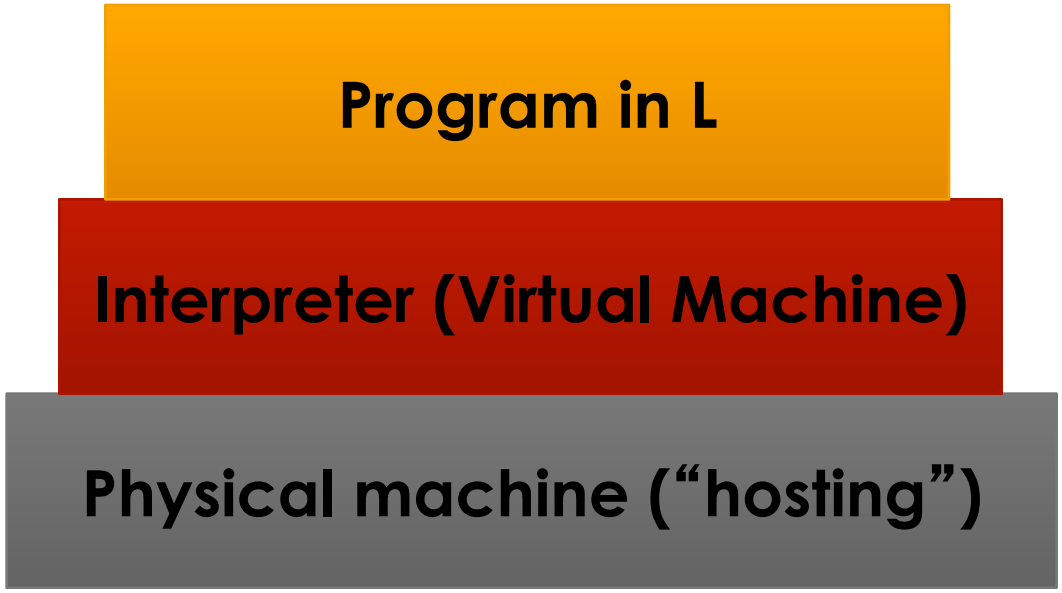
Lecture 5

Formal Languages and Compilers 2011

Nataliia Bielova

Definition

- Intepreter 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

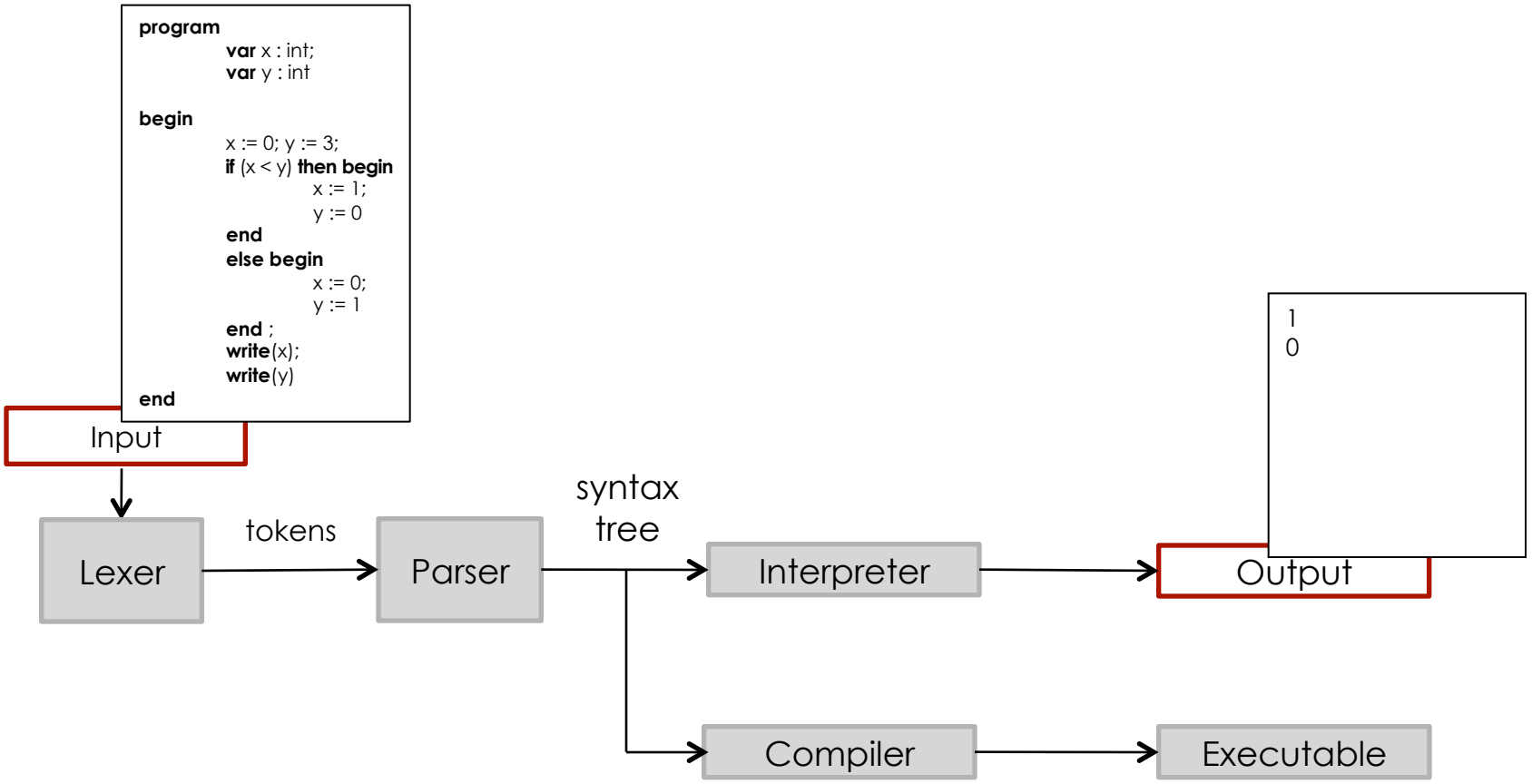
```
program
    var x : int;
    var y : int

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 ⇒

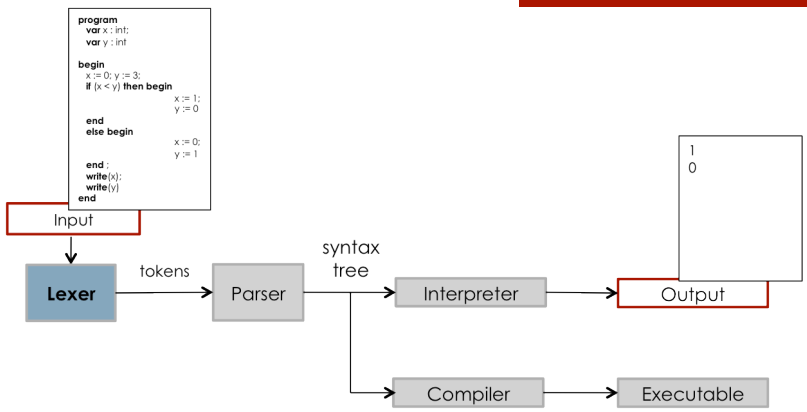
```
1
0
```

Interpreter or compiler?



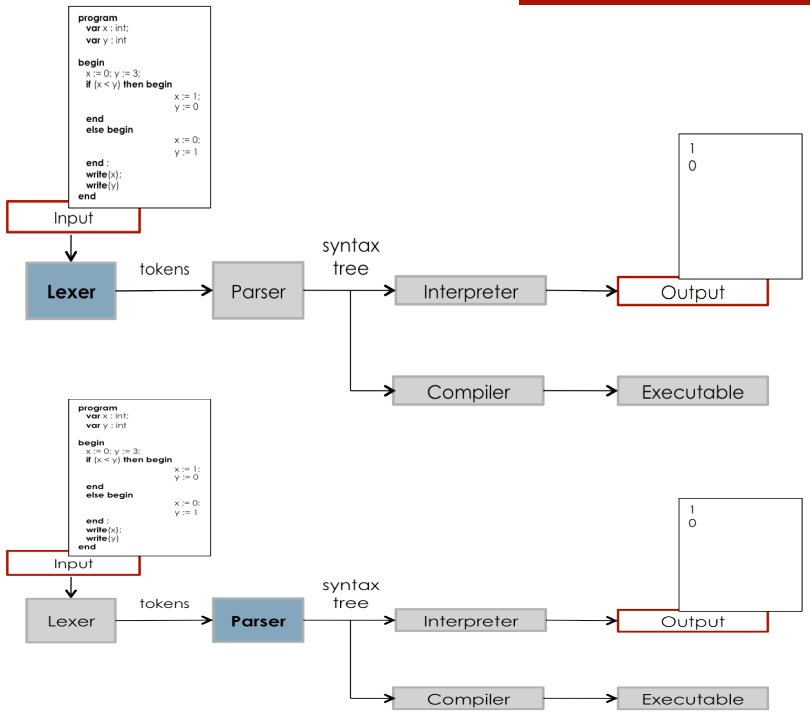
Elements of interpreter

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



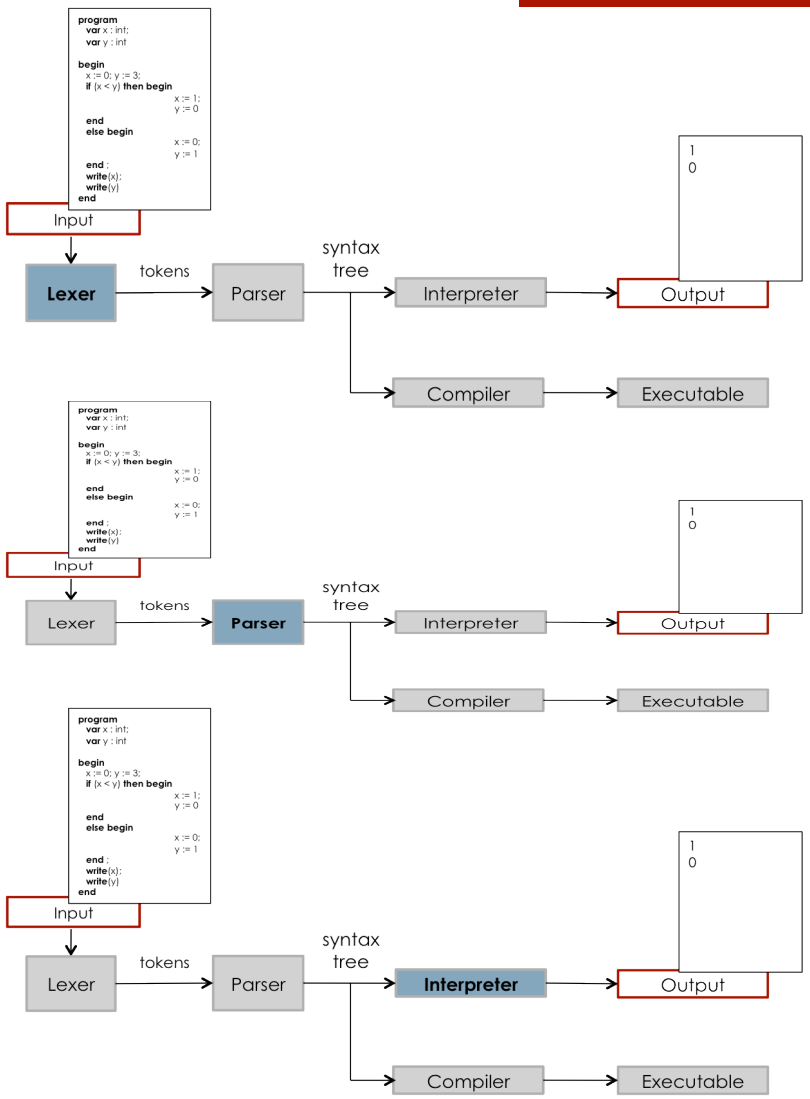
Elements of interpreter

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



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

t 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

  $Env : Id -> (Loc \cup Val)$ 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

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

$$E\llbracket \mathrm{Sum}(\mathrm{n}_1, \mathrm{n}_2) \rrbracket_{\mathrm{rs}} = E\llbracket \mathrm{n}_1 \rrbracket_{\mathrm{rs}} + E\llbracket \mathrm{n}_2 \rrbracket_{\mathrm{rs}}$$

$$E\llbracket \mathrm{i} \rrbracket_{\mathrm{rs}} = \begin{cases} \mathrm{s}(\mathrm{r}(\mathrm{i})) & \text{if} \quad \mathrm{r}(\mathrm{i}) \in Loc \\ \mathrm{r}(\mathrm{i}) & \text{if} \quad \mathrm{r}(\mathrm{i}) \in Val \end{cases}$$

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

$$B\llbracket \mathrm{Or}(\mathrm{b}_1, \mathrm{b}_2) \rrbracket_{\mathrm{rs}} = \begin{cases} \mathrm{true} & \text{if } B\llbracket b_1 \rrbracket_{rs} \text{ is true} \\ B\llbracket b_2 \rrbracket_{rs} & \text{otherwise} \end{cases}$$

Declaration: evaluation

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

$$D\llbracket \text{const v} = \text{n} \rrbracket_{rs} = \text{r's}$$

where:

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

$$D\llbracket \text{var v} := \text{n} \rrbracket_{rs} = \text{r's'}$$

where:

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

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

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

Commands: execution

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

$$C[\![\text{X} := \text{e}]\!]_{rs} = \text{s}'$$

where:

$$\text{l} = \Lambda[\![\text{X}]\!]_{rs}$$

$$\text{v} = E[\![\text{e}]\!]_{rs}$$

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

$$C[\![\text{if b then } \text{c}_1 \text{ else } \text{c}_2]\!]_{rs} = \text{s}'$$

where:

$$\text{s}' = \begin{cases} C[\![\text{c}_1]\!]_{rs} & \text{if } B[\![b]\!]_{rs} = \text{true} \\ C[\![\text{c}_2]\!]_{rs} & \text{otherwise} \end{cases}$$

$$C[\![\text{while b do c}]\!]_{rs} = \begin{cases} \text{s} & \text{if } B[\![b]\!]_{rs} = \text{false} \\ C[\![\text{while b do c}]\!]_{rs''} & \text{otherwise} \end{cases}$$

where $\text{s}'' = C[\![\text{c}]\!]_{rs}$

Example: repeat - until

repeat
  cmd
until bexp

$$C \llbracket \text{repeat cmd until bexp} \rrbracket_{rs} = s'$$

where :

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

$$s'' = C \llbracket \text{cmd} \rrbracket_{rs}$$

Example: repeat - until

```
repeat
   cmd
until  bexp
```

- parser.mly: token REPEAT and UNTIL
- lexer.mll: strings ”repeat” and ”until”
- syntaxtree.ml: contructor 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:

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

- Factorial of the number:

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