Functional Programming
Lecture 5

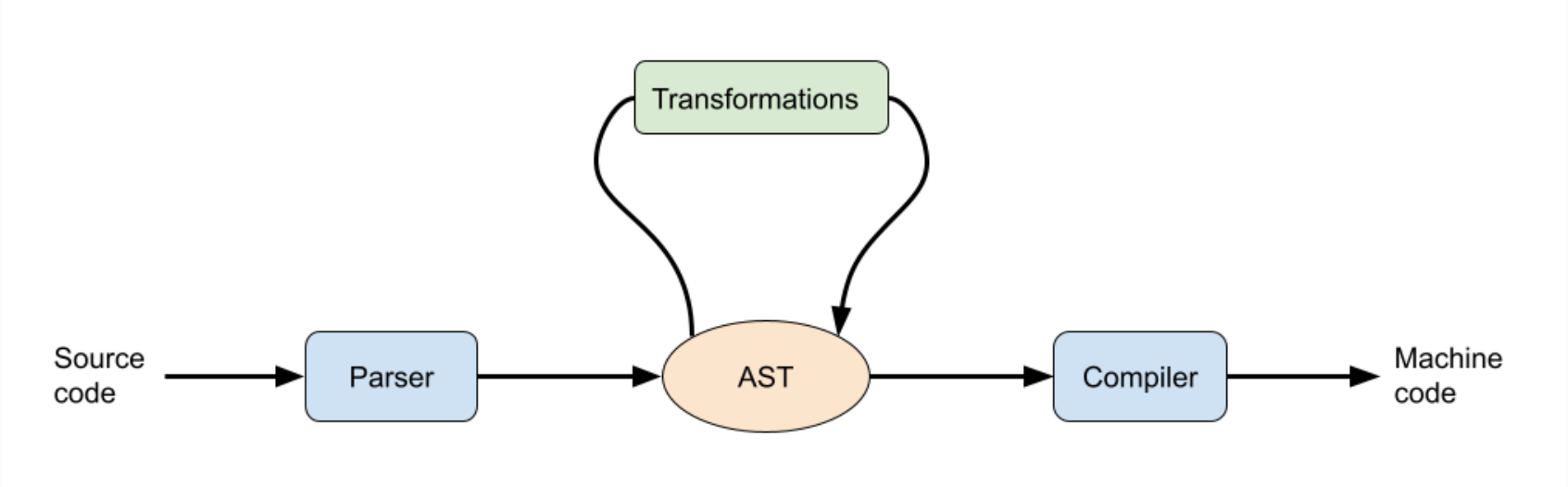
Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Syntax macros

Syntax macros

Racket macro system is a powerful tool allowing to extend syntax. It operates on AST not source code.



Example

```
(define-syntax macro-if
  (syntax-rules ()
    [(macro-if c a b)
     (my-lazy-if c (thunk a) (thunk b))]))

(macro-if (null? '()) '() (car '()))
```

List comprehension

```
[x+2 for x in [2,3,5] if 3 >= x]
```

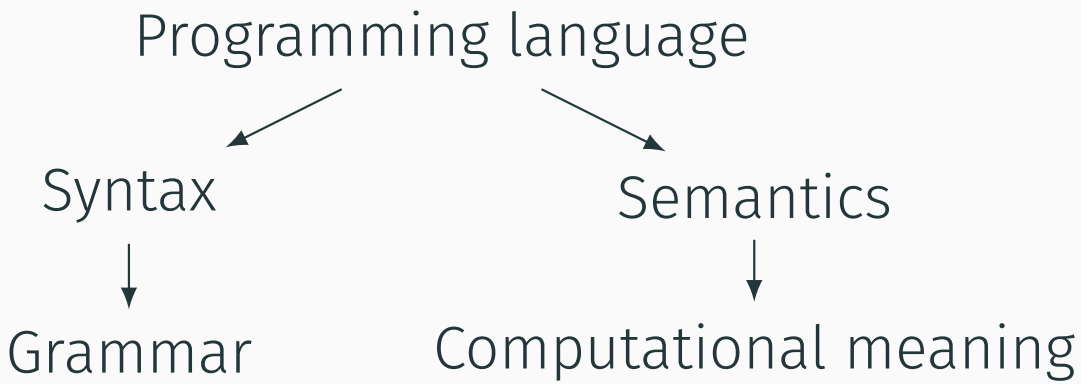
```
(define-syntax list-comp
  (syntax-rules (: <- if)
    [(list-comp <expr> : <id> <- <lst>)
     (map (lambda (<id>) <expr>) <lst>)]

    [(list-comp <expr> : <id> <- <lst> if <cond>)
     (map (lambda (<id>) <expr>)
          (filter (lambda (<id>) <cond>) <lst>))]))

(list-comp (+ x 2) : x <- '(2 3 5) if (>= 3 x))
```

Interpreters

Programming languages and interpreters



To implement an interpreter of a LISP-like language in Scheme/Racket no parser is needed. We can use the built-in parser.

Brainf*ck syntax

A program to add two numbers:

```
,>,[-<+>]<.
```

Brainf*ck syntax

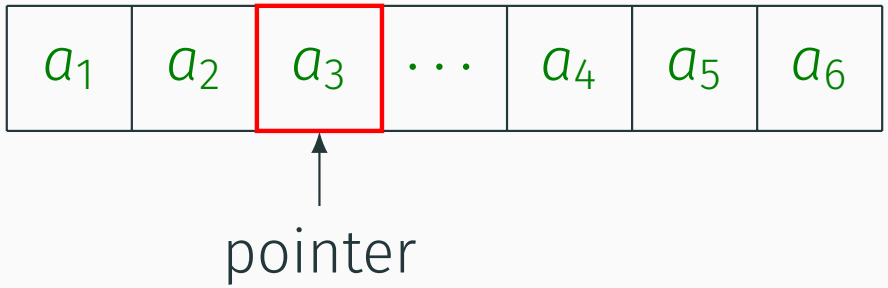
Brainf*ck is a minimalistic language defining computations over a fixed-size tape of numbers.

```
<program> -> <term>*
   <term> -> <cmd> | <cycle>
  <cycle> -> [<program>]
    <cmd> -> + | - | < | > | . | ,
```

Example of a syntactically correct Brainf*ck program:
`,>,[-<+>]<.`

Brainf*ck semantics

State of computation is captured by a fixed-size tape of numbers initialized by 0s.



`+`	`-`	increase/decrease focused number
`<`	`>`	move pointer left/right
`.`	`,`	output/input focused number
`[`	`]`	while focused number isn't 0, execute inner code

Brainf*ck interpreter in Racket

We represent Brainf*ck programs as lists of symbols.

Cycles form nested lists.

Symbols `.` and `,` are replaced by `*` and `@` respectively.

```
(define add-prg '(@ > @ [- < + >] < *))
```

Tape is represented by a mutable vector of numbers.

```
(run-prg add-prg '(12 34))
```

displays `46 done`.

Homework assignment 2 - SVGen interpreter

Aim: Try to implement a simple interpreter

SVGen is a simple programming language for generating SVG images.

Points: 15

Deadline: in 3 weeks (April 8th)

Penalty: after deadline -1 points every day (at most -14)

Description: all details can be found in CW

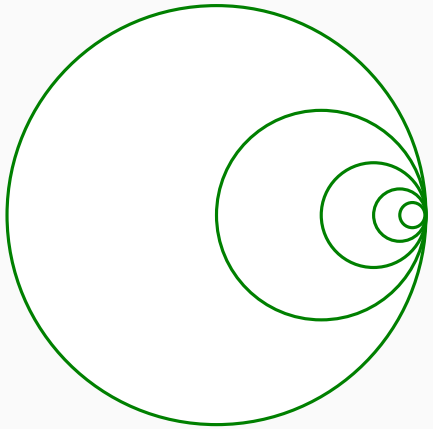
The use of buit-in function `eval` is not allowed!

Make your code pure!

SVGen example

```
(define test2
  '((define STYLE "fill:white;stroke:green;stroke-width:3")
    (define (circles x r)
      (when (> r 10)
        (circle x 200 r STYLE)
        (circles (+ x (floor (/ r 2))) (floor (/ r 2)))))))

(execute 400 400 test2 '(circles 200 200))
```



Handling function calls

During the evaluation, one must maintain an environment — a data structure consisting of two parts:

1. a structure containing definitions
2. and variable bindings.

To evaluate `(fn exp1 exp2 ...)`, do

1. Eval numeric expressions `exp1,exp2,...` obtaining `val1,val2,...`
2. Create a new environment containing variable bindings for arguments of `fn` using values `val1,val2,...`
3. Evaluate the sequence of expressions in the body of `fn`

What have we learned?

- Syntax can be extended by macros operating on AST.
- Programming language is determined by its syntax and semantics.
- Using syntax, interpreter parses source code into Abstract Syntax Tree
- and then evaluates/executes the program based on semantics.