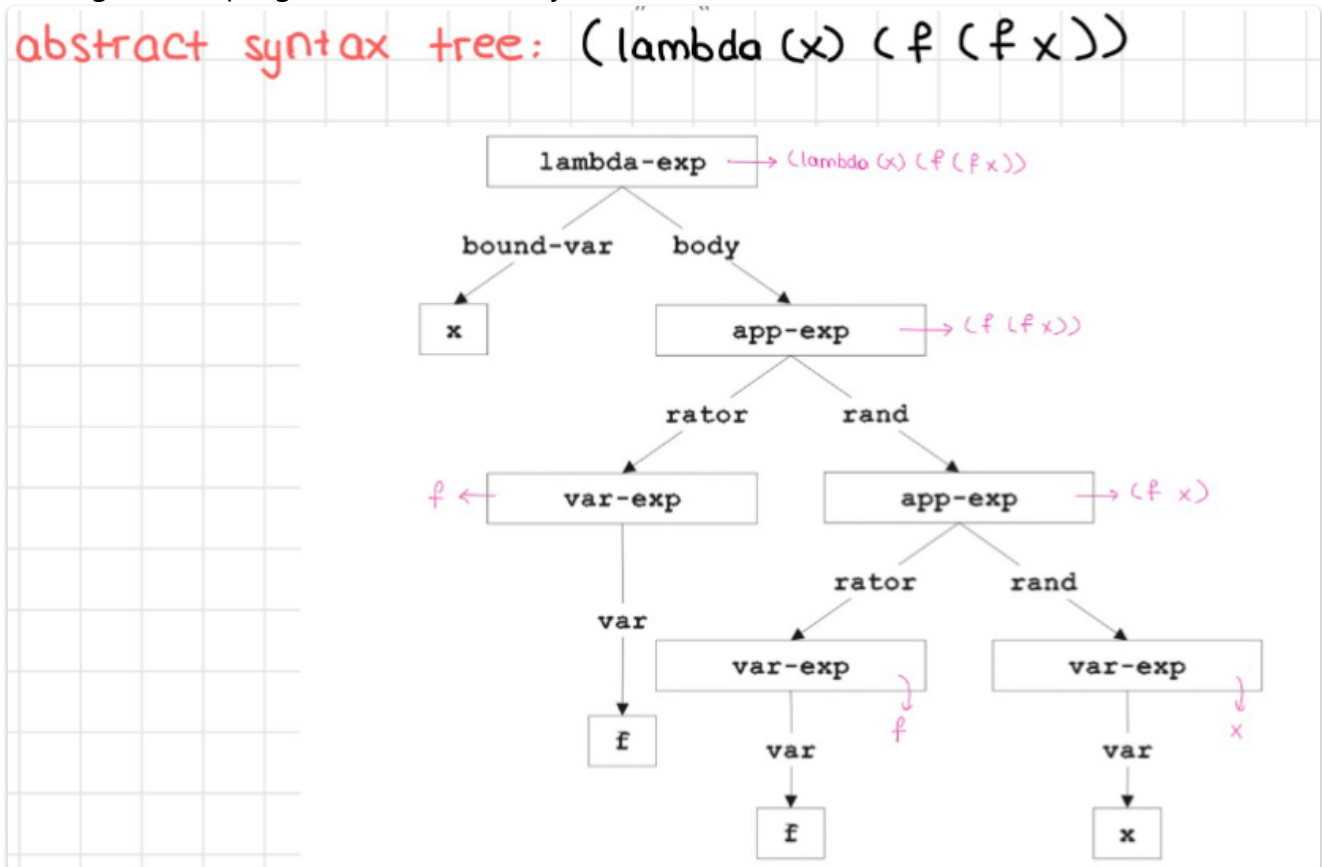


10. Abstract Syntax, Representation, Interpretation

[lecture 10 -- Syntax Representation & Interpreters.pdf](#)

Abstract Syntax Tree

Parsing takes a program and builds a syntax tree.



Parsing and Unparsing

parsing: text file \rightarrow syntax tree

unparsing: syntax tree \rightarrow text file

$LcExp ::= Identifier$

$::= (\text{lambda } (Identifier) LcExp)$

$::= (LcExp LcExp)$

$\text{parse-expression} : SchemeVal \rightarrow LcExp$

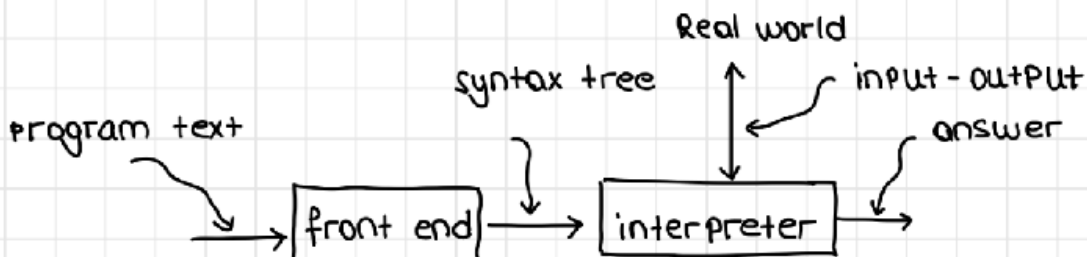
```
(define parse-expression
  (lambda (datum)
    (cond
      ((symbol? datum) (var-exp datum))
      ((pair? datum)
       (if (eqv? (car datum) 'lambda)
           (lambda-exp
            (car (cadr datum))
            (parse-expression (caddr datum)))
           (app-exp
            (parse-expression (car datum))
            (parse-expression (cadr datum)))))
      (else (report-invalid-concrete-syntax datum))))
```

$\text{unparse-lc-exp} : LcExp \rightarrow SchemeVal$

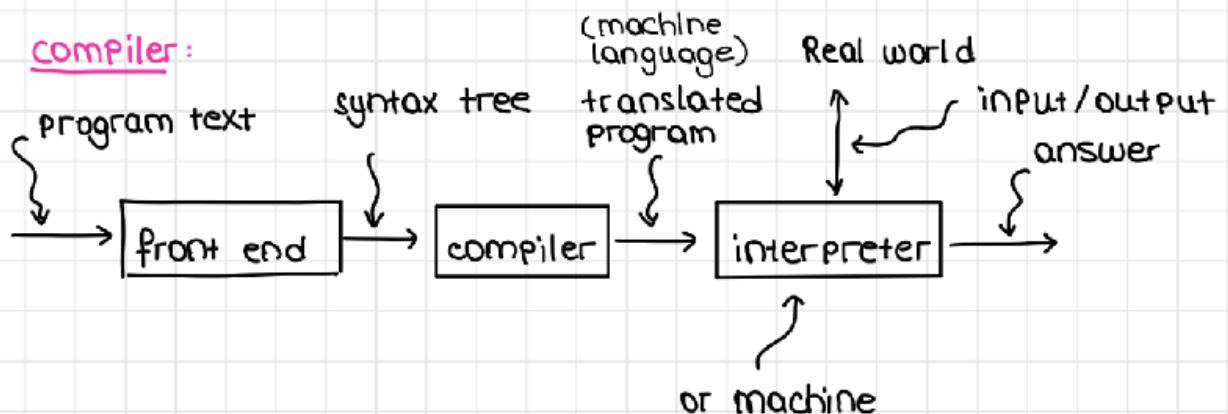
```
(define unparse-lc-exp
  (lambda (exp)
    (cases lc-exp exp
      (var-exp (var) var)
      (lambda-exp (bound-var body)
        (list 'lambda (list bound-var)
              (unparse-lc-exp body)))
      (app-exp (rator rand)
        (list (unparse-lc-exp rator)
              (unparse-lc-exp rand))))))
```

Compilers

interpreter:



compiler:



Let Language

LET Language:

grammar

Program ::= *Expression* → concrete syntax
a-program (exp1) → abstract syntax

Expression ::= *Number*
const-exp (num)

Expression ::= -(*Expression* , *Expression*)
diff-exp (exp1 exp2)

Expression ::= zero? (*Expression*)
zero?-exp (exp1)

Expression ::= if *Expression* then *Expression* else *Expression*
if-exp (exp1 exp2 exp3)

Expression ::= *Identifier*
var-exp (var)

Expression ::= let x = 5 -(x, 5) *Expression* in *Expression*
let-exp (var exp1 body)