# CSE 341
# Lecture 19

parsing / Homework 7

slides created by Marty Stepp

http://www.cs.washington.edu/341/

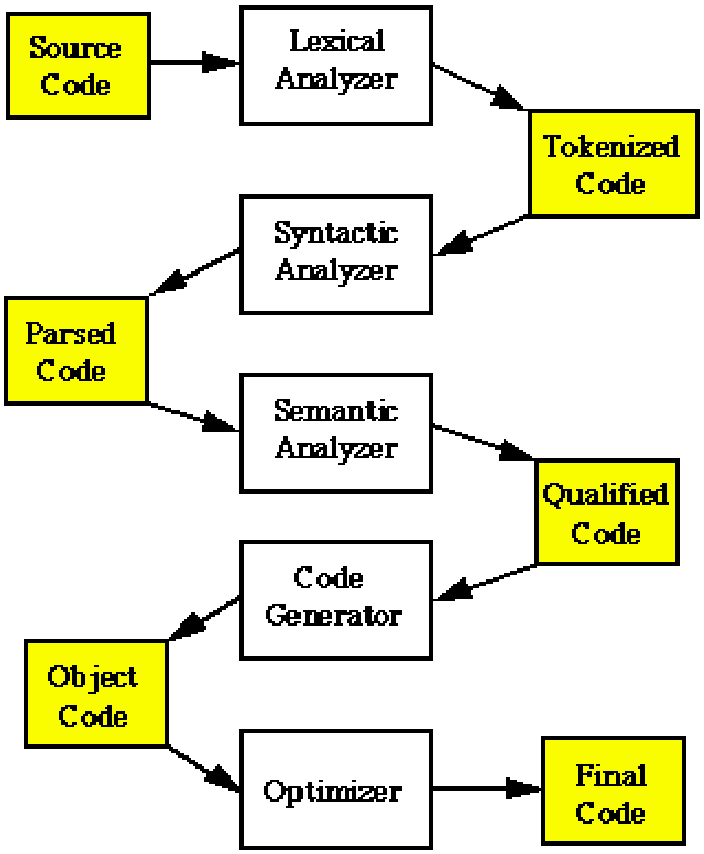
# Looking ahead

- We will complete a 2-part assignment related to analyzing and interpreting BASIC source code.
  - **HW7**: BASIC expression parser
  - **HW8**: BASIC interpreter

- To complete this assignment, it is helpful to have some background about how compilers and interpreters work.
  - HW8 will be an interpreter that performs REPL (read, eval, print) on BASIC source code.
  - HW7 is a parser that reads BASIC math expressions.
    - HW8 will make use of HW7's code to eval expressions.

# How does a compiler work?

- A typical compiler or interpreter consists of many steps:
  1. **lexical analysis:** break apart the code into tokens
  2. **syntax analysis (parsing)**: examine sequences of tokens based on the language's syntax
  3. **semantic analysis**: reason about the meaning of the token sequences (particularly pertaining to types)
  4. **code generation**: generate executable code in some format (native, bytecode, etc.)
  5. **optimization** (optional): improve the generated code



# 1. Lexical analysis (tokenizing)

- Suppose you are writing a Java interpreter or compiler.
  - The source code you want to read contains this:

```
for (int i=2*3/4 + 2+7; i*x <= 3.7 * y; i = i*3+7)
```

  - The first task is to split apart the input into *tokens* based on the language's token syntax and delimiters:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `for` | `(` | `int` | `i` | `=` | `2` | `*` | `3` | `/` | `4` | `+` | `2` | `+` | `7` | `;` | `i` |
| `*` | `x` | `<=` | `3.7` | `*` | `y` | `;` | `i` | `=` | `i` | `*` | `3` | `+` | `7` | `)` | |

# A tokenizer in Scheme

- If our Java interpreter is written in Scheme, we convert:

```
for (int i=2*3/4 + 2+7; i*x <= 3.7 * y; i = i*3+7)
```

  - into the following Scheme list:

```
(for ( int i = 2 * 3 / 4 + 2 + 7 ; i * x <= 3.7 * y ; i =
i * 3 + 7 ) )
```

    – *if typed in as Scheme source, it would have been:*

```
(list 'for '( 'int 'i '= 2 '* 3 '/ 4 '+ 2 '+ 7 '; 'i '* 'x
'<= 3.7 '* 'y '; 'i '= 'i '* 3 '+ 7 ') )
```

  - ( and ) are hard to process as symbols; so we'll use:

```
(for lparen int i = 2 * 3 / 4 + 2 + 7 ; i * x <= 3.7 * y
; i = i * 3 + 7 rparen )
```

# 2. Syntax analysis (parsing)

- Now that we have a list of tokens, we will walk across that list to see how the tokens relate to each other.
  - Example: Suppose we've processed the source code up to:

```
(for lparen int i = 2 * 3 / 4 + 2 + 7 ; i * x <= 3.7 * y
                    ^
; i = i * 3 + 7 rparen )
```

  - From parser's perspective, the list of upcoming tokens is:

```
2 * 3 / 4 + 2 + 7 ; i * x <= 3.7 * y ; i = ...
^
```

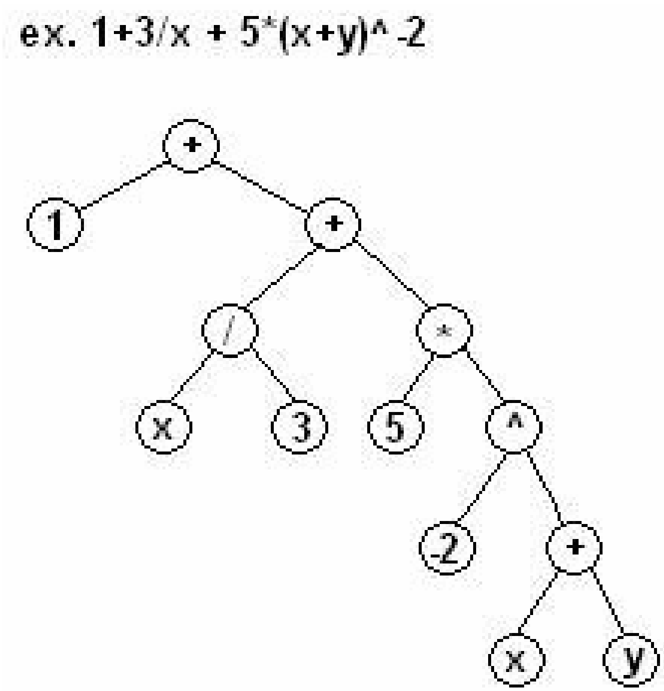
# Parsing expressions

- The list of upcoming tokens contains expressions:

```
2 * 3 / 4 + 2 + 7 ; i * x <= 3.7 * y ; i = ...
```

- Parsers process the code they read:
  - a compiler builds a *syntax tree*
  - an interpreter *evaluates* the code

```
10 ; i * x <= 3.7 * y ; i = ...
```



# Grammars

- *<test>* ::= *<expr>* *<relop>* *<expr>*
- *<relop>* ::= "<" | ">" | "<=" | ">=" | "=" | "<>"
- *<expr>* ::= *<term>* {("+" | "-") *<term>*}
- *<term>* ::= *<element>* {("*" | "/") *<element>*}
- *<element>* ::= *<factor>* {"^" *<factor>*}
- *<factor>* ::= *<number>* | ("+" | "-") *<factor>* | "(" *<expr>* ")"
  | *<f>* "(" *<expr>* ")"
- *<f>* ::= SIN | COS | TAN | ATN | EXP | ABS | LOG | SQR | RND | INT

- **grammar:** set of structural rules for a language
  - often described in terms of themselves (recursive)
    - *<non-terminal>*; TERMINAL; "literal token";
    - {repeated 0--* times}; or: (a | b)

# Procedures you'll write (1)

- `parse-factor`
  - *<factor>* ::= *<number>* | ("+" | "-") *<factor>* | "(" *<expr>* ")"
    | *<f>* "(" *<expr>* ")"

```
> (parse-factor '(- 7.9 3.4 * 7.2))
(-7.9 3.4 * 7.2)

> (parse-factor '(lparen 7.3 - 3.4 rparen + 3.4))
(3.9 + 3.4)

> (parse-factor '(SQR lparen 12 + 3 * 6 - 5 rparen))
(5)

> (parse-factor '(- lparen 2 + 2 rparen * 4.5))
(-4 * 4.5)
```

# Procedures you'll write (2)

- parse-element
  - *<element>* ::= *<factor>* {"^" *<factor>*}

```
> (parse-element '(2 ^ 2 ^ 3 THEN 450))
(64 THEN 450)

> (parse-element '(2 ^ 2 ^ -3 THEN 450))
(0.015625 THEN 450

> (parse-element '(2.3 ^ 4.5 * 7.3))
(42.43998894277659 * 7.3)

> (parse-element '(7.4 + 2.3))
(7.4 + 2.3)
```

# The grammar is the code!

- *<factor>* ::= ***<number>*** | **("+" | "-") *<factor>*** | **"(" *<expr>* ")"**
  | ***<f>* "(" *<expr>* ")"**

```
(define (parse-factor lst)
    ; 1) if I see a number, then ...
    ; 2) if I see a + or -, then ...
    ; 3) if I see a (,      then ...
    ; 4) else it is an <f>, so   ...
```

- How do you know which of the four cases you are in?

# Recall: Checking types

(***type***? ***expr***)

- tests whether the expression/var is of the given type
  - `(integer? 42)` → `#t`
  - `(rational? 3/4)` → `#t`
  - `(real? 42.4)` → `#t`
  - `(number? 42)` → `#t`
  - `(procedure? +)` → `#t`
  - `(string? "hi")` → `#t`
  - `(symbol? 'a)` → `#t`
  - `(list? '(1 2 3))` → `#t`
  - `(pair? (42 . 17))` → `#t`

# Exact vs. inexact numbers

- You'll encounter problems with Scheme's rational type:
  - Scheme thinks 3/2 is 1 $^{1}/_{2}$ (a rational)
  - the interpreter wants 3/2 to be 1.5 (a real)

- Scheme differentiates *exact* numbers (integers, fractions) from *inexact* numbers (real numbers).
  - (A complex number can be exact or inexact.)
  - Round-off errors can occur only with inexact numbers.

# Managing exact/inexact numbers

- `exact?`, `inexact?` procedures see if a number is exact:
  - `(exact? 42)` → `#t`
  - `(inexact? 3.25)` → `#t`

- Scheme has procedures to **convert** between the two:
  - `(exact->inexact 13/4)` → `3.25`
  - `(inexact->exact 3.25)` → `3 1/4`
    - (May want `floor`, `ceiling`, `truncate`, ... in some cases.)

(In general, conversion procedure names are ***`type1->type2`*** .)

# Parsing math functions

- <*f*> ::= SIN | COS | TAN | ATN | EXP | ABS | LOG | SQR | RND | INT
- grammar has tokens representing various math functions
  - must map from these to equivalent Scheme procedures
  - could use a giant nested `if` or `cond` expression, but...

```
(define functions
  '((SIN . sin)  (COS . cos) (TAN . tan) (ATN . atan)
    (EXP . exp)  (ABS . abs) (LOG . log) (SQR . sqrt)
    (RND . rand) (INT . trunc)))
```

# Associative lists (maps) with pairs

- Recall: a **map** associates *keys* with *values*
  - can retrieve a value later by supplying the key
- in Scheme, a map is stored as a list of key/value **pairs**:

```
(define phonebook (list '(Marty 6852181)
     '(Stuart 6859138) '(Jenny 8675309)))
```

- look things up in a map using the `assoc` procedure:

```
> (assoc 'Stuart phonebook)
(Stuart 6859138)
> (cdr (assoc 'Jenny phonebook))  ; get value
8675309
```