# Comp 411
# Principles of Programming Languages
# Lecture 2
# Syntax

Corky Cartwright

August 24, 2011

# Syntax: The Boring Part of Programming Languages

- Programs are represented by sequences of symbols.
- These symbols are represented as sequences of characters that can be typed on a keyboard (ASCII).
- What about Unicode?
- To analyze or execute the programs written in a language, we must translate the ASCII representation for a program to a higher-level tree representation. This process, called *parsing*, conveniently breaks into two parts:
  - *lexical analysis*, and
  - *context-free parsing* (often simply called *parsing*).

# Lexical Analysis

- Consider this sequence of characters: `begin middle end`
- What are the smallest meaningful pieces of syntax in this phrase?
- The process of converting a character stream into a corresponding sequence of meaningful symbols (called *tokens* or *lexemes*) is called *tokenizing*, *lexing* or *lexical analysis*. A program that performs this process is called a *tokenizer, lexer,* or *scanner*.
- In Scheme, we tokenize `(set! x (+ x 1))` as

```
(   set!    x    (    +    x    1    )    )
```

- Similarly, in Java, we tokenize

  `System.out.println("Hello World!");` as

```
System  .  out .  println  (  "Hello World!"  )  ;
```

# Lexical Analysis, cont.

- Tokenizing is straightforward for most languages because it can be performed by a finite automaton [regular grammar] (Fortran is an exception!).
  - The rules governing this process are (a very boring) part of the language definition.
- Parsing a stream of tokens into structural description of a program (typically a tree) is harder.

# Parsing

- Consider the Java statement: `x = x + 1;` where `x` is an `int` variable.
- The grammar for Java stipulates (among other things):
  - The assignment operator `=` may be preceded by an identifier and must be followed by an expression.
  - An expression may be two expressions separated by a binary operator, such as `+`.
  - An assignment expression can serve as a statement if it is followed by the terminator symbol `;`.

Given all of the rules of this grammar, we can deduce that the sequence of characters (tokens)

```
x = x + 1;
```

is a legal program statement.

# Parsing Token Streams into Trees

- Consider the following ways to express an assignment operation:

```
x = x + 1
x := x + 1
(set! x (+ x 1))
```

- Which of these do you prefer?
- It should not matter very much.
- To eliminate the irrelevant syntactic details, we can create a data representation that formulates program syntax as trees. For instance, the abstract syntax for the assignment code given above could be

```
(make-assignment <Rep of x> <Rep of x + 1>)
```

- or

```
new Assignment(<Rep of x> , <Rep of x + 1>)
```

# A Simple Example

```
Exp ::= Num | Var | (Exp Exp) | (lambda Var Exp)
```

**Num** is the set of numeric constants (given in the lexer specification)
**Var** is the set of variable names (given in the lexer specification)

- To represent this syntax as trees (abstract syntax) in Scheme

```
; exp := (make-num number) + (make-var symbol) + (make-app exp exp) +
;             (make-proc symbol exp)
(define-struct (num n))
(define-struct (var s))
(define-struct (app rator rand))
(define-struct (proc param body))  ;; param is a symbol not a var
```

  **app** represents a function application
  **proc** represents a function definition

- In Java, we represent the same data definition using the composite pattern. In Scala, there is a special form of class (called a variant) for representing functional data.

# Top Down (Predictive) Parsing

Idea: design the grammar so that we can always tell what rule to use next starting from the root of the parse tree by looking ahead some small number [k] of tokens (formalized as LL(k) parsing).

Can easily be implemented by hand by writing one recursive procedure for each syntactic category (non-terminal symbol). The code in each procedure matches the token pattern of the right hand side of the rule for that procedure against the token stream. This approach to writing a parser is called *recursive descent.*

Conceptual aid: syntax diagrams to express context free grammars.

Recursive descent and syntax diagrams are discussed in next lecture.