Comp 411
Principles of Programming Languages
Lecture 2
Syntax

Corky Cartwright
January 11, 2017
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? (Potentially important in practice.)
- To analyze or execute the programs written in a language, we must translate the ASCII/Unicode 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* or a *lexer*.

- In Scheme, we tokenize \((\text{set! } x (+ x 1))\) as
  \[
  (\text{set! } x ( + x 1 ))
  \]

- Similarly, in Java, we tokenize

\[
\text{System.out.println("Hello World!"); as}
\text{System . out . println ("Hello World!");}
\]
Lexical Analysis, cont.

- Tokenizing is straightforward for most languages because it can be performed by a finite automaton (equivalent to a regular grammar for those of you who have taken 412 or 481) that matches the longest possible string of characters as the next token. Fortran is an interesting exception!.
  - The rules governing this process are (a very boring) part of the language definition.
  - The details are generally provided as part of a language definition but subsequently glossed over as uninteresting.
- Parsing a stream of tokens into structural description of a program (typically a tree) is harder.
• Consider the Java statement: \( \text{x} = \text{x} + 1; \)
  where \( \text{x} \) is an \textbf{int} variable.

• The grammar for Java stipulates (among other things):
  – The assignment operator may be preceded by an identifier (other more complex, possibilities exist as well) and must be followed by an expression.
  – An expression may be two expressions (technically restricted to special kinds of expressions) separated by a binary operator, such as \(+\).
  – An assignment expression can serve as a statement if it is followed by the statement terminator symbol \( ; \).

Hence, we can deduce from the grammatical rules of Java that the above sequence of characters (tokens) is a legal program statement.
Parsing Token Streams into Trees

• Consider the following ways to express an assignment operation:

\[
\begin{align*}
\text{x} &= \text{x} + 1 & \text{[Java]} \\
\text{x} &:= \text{x} + 1 & \text{[Algol]} \\
(\text{set! x \,(+\ x\ 1)}) & & \text{[Scheme]}
\end{align*}
\]

• Which of these do you prefer?
• It should not matter 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 (assuming Scheme as the implementation language)

\[
(\text{make-assignment} \,<\text{Rep of x}\,>\, \,<\text{Rep of x}+1\,>)
\]

• Or (in Java as the implementation language)

\[
\text{new Assignment}(\,<\text{Rep of x}\,\,>\,\,\,\,<\text{Rep of x}\, +\ 1\,>)
\]
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!

where an app structure represents a function application and a proc structure represents a function definition
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 (LL(k) parsing).

This algorithm an easily be implemented by hand: called recursive descent.

Conceptual aid: syntax diagrams to express context free grammars.

Intuition: k-symbol look-ahead is used to determine which branch to take at a fork in a syntax diagram.

We try to design LL(k) grammars so that k is 1.