Comp 411
Principles of Programming Languages
Lecture 3
Syntax

Corky Cartwright

August 29, 2012

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? A messy, evolving standard. Most applications assume ASCII. Some special purpose hacks are required to deal with other languages/keyboards based on the Roman alphabet.
- 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 Scala
-

```
/* Exp := Number + Variable + App + Proc
sealed trait Exp
case class Number(Int n) extends Exp
case class Variable(Symbol s) extends Exp
case class App(rator: Exp, rand: Exp) extends Exp
case class Proc(param: String, body: Exp)
// Note: param is a Symbol not a Variable.  Symbol is a built-in type.
```

App represents a function application
Proc represents a function definition

- Note that we have represented this abstract syntax in Scala using the composite pattern.

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.

Top Down Parsing Details

- What is a context-free grammar (CFG)?
  A recursive definition of a set of strings; it is *identical* in format to the data definitions used in HTDP/EOOD *except* for the fact that it defines sets of strings (using concatenation) rather than sets of trees (objects/structs) using tree construction. The *root symbol* of a grammar generates the language of he grammar. In other words, it designates the syntax of complete programs.
- Example. The language of expressions generated by `<expr>`

```
<expr> ::= <term>  |  <term> + <expr>
<term> ::= <number>  |  <variable>  |  ( <expr> )
```

- Some sample strings generated by this CFG

```
5      5+10      5+10+7      (5+10)+7
```

- What is the fundamental difference between generating strings and generating trees?
  - The derivation of a generated tree is manifest in the structure of the tree.
  - The derivation of a generated string is *not* manifest in the structure of the string; it must be *reconstructed* by the parsing process. This reconstruction may be *amibiguous* and it may be costly in the general case (O(n^3)). Fortunately, *deterministic* (LR(*k*)) parsing is linear.

Top Down Parsing cont.

- Data definition corresponding to sample grammar:

```
Expr  = Expr + Expr  |  Number  |  Variable
```

- Why is the data definition simpler? (Why did we introduce the syntactic category `<term>` in the CFG?)

- Consider the following example:

```
5+10+7
```

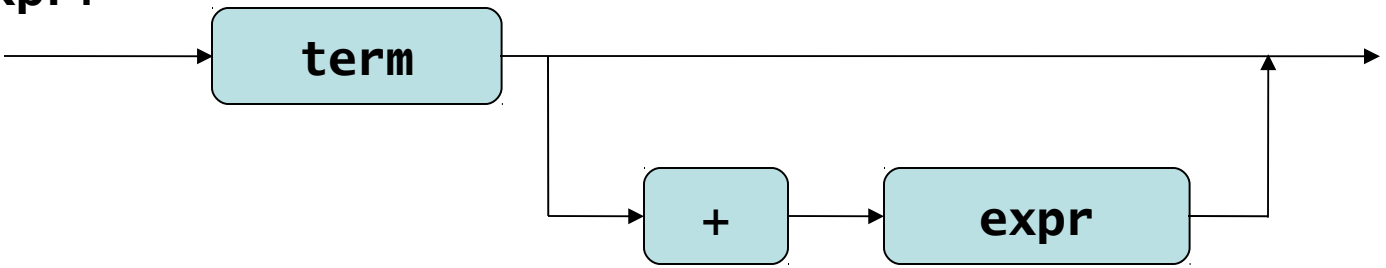
- Are strings a good data representation for programs in software?

- Why do we use string representations for programs? In what contexts is this a good representation?

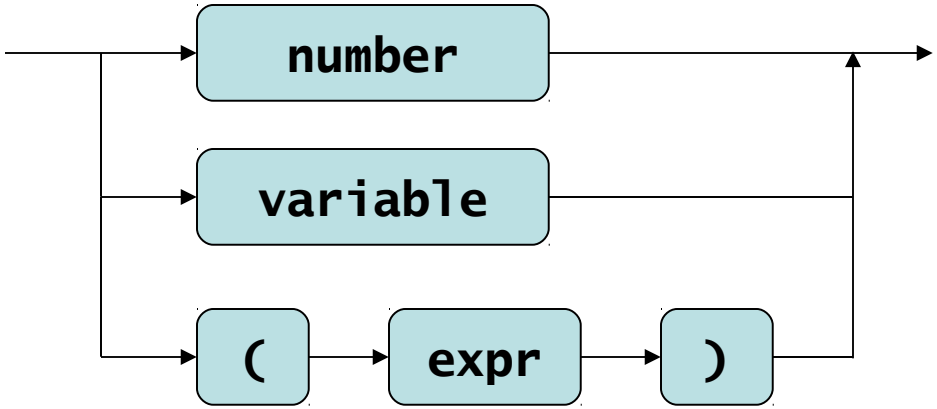
Parsing algorithms

- Top-down (predictive) parsing: use *k* token look-ahead to determine next syntactic category.
- Simplest description uses *syntax diagrams* (intuition: railroad tracks, plumbing)

`expr`:



`term`:



Key Idea in Top Down Parsing

- Use *k* token look-ahead to determine which direction to go at a branch point in the current syntax diagram.
- Example: **5+10**
  - Start parsing by reading first token **5** and matching the syntax diagram for **expr**
  - Must recognize a **term**; invoke rule (diagram) for **term**
  - Select the **number** branch (path) based on current token **5**
  - Digest the current token to match **number** and read next token **+**; return from **term** back to **expr**
  - Select the **+** branch in **expr** diagram based on current token
  - Digest the current token to match **+** and read the next token **10**
  - Must recognize an **expr**; invoke rule (diagram) for **expr**
  - Must recognize a **term**; invoke rule (diagram) for **term**
  - Select the **number** branch based on current token **10**
  - Digest the current token to match **number** and read next token **EOF**
  - Return from **term**; return from **expr**

Designing Grammars for Top-Down Parsing

- Many different grammars generate the same language (set of strings):
- Requirement for any efficient parsing technique: determinism (non-ambiguity)
- For deterministic *top-down* parsing, we must 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).
- For top down parsing
  - Eliminate left recursion; use right recursion instead
  - Factor out common prefixes (as in syntax diagrams)
  - Use iteration in syntax diagrams instead of right recursion where necessary
  - In extreme cases, hack the lexer to split token categories based on local context

Other Parsing Methods

When we parse a sentence using a CFG, we effectively build a (parse) tree showing how to construct the sentence using the grammar. The root (start) symbol is the root of the tree and the tokens in the input stream are the leaves.

Top-down (predictive) parsing is simple and intuitive, but is is not as powerful a deterministic parsing strategy as bottom-up parsing which is much more tedious. Bottom up deterministic parsing is formalized as LR(*k*) parsing.
Every LL(*k*) grammar is also LR(1) but many LR(1) grammars are not LL(*k*) for any *k*.

No sane person manually writes a bottom-up parser. In other words, there is no credible bottom-up alternative to recursive descent parsing. Bottom-up parsers are generated using parser-generator tools which until recently were almost universally based on LR(k) parsing (or some bottom-up restriction of LR(k) such as SLR(k) or LALR(k)). But some newer parser generators like javacc are based on LL(k) parsing. In DrJava, we have several different parsers including both recursive descent parsers and automatically generated parsers produced by javacc.

Why is top-down parsing making inroads among parser generators? Top-down parsing is much easier to understand and more amenable to generating intelligible syntax diagnostics. Why is recursive descent still used in production compilers? Because it is straightforward (if tedious) to code, supports sensible error diagnostics, and accommodates *ad hoc* hacks (e.g., use of state).to get around the LL(k) restriction.

If you want to learn about the mechanics of bottom-up parsing, take Comp 412.

Other Parsing Methods

General frameworks for parsing (driven by arbitrary CFGs as input) must accommodate non-deterministic languages and perform some back-tracking.

A decade ago, this approach was much too slow to warrant serious consideration, but contemporary machines are so fact that general frameworks may be a good choice for building parsers for applications where programs of modest size are expected, e.g. most domain specific languages. Does such a parser make sense for a production compiler for a language like Java, Scala, C, C++, Python, etc.? No! (But it might partially explain why the Scala compiler is so slow. Seriously, I suspect that Scala uses some form of deterministic parser.)