Class 15: Parsing Expressions

Held: Wednesday, 28 September 2011

Summary: We consider a grammar for expressions.

Related Pages:
- EBoard.
- Reading: Aho et al., 4.3.

Notes:
- EC for Thursday’s CS extra.

Overview:
- BNF Grammars, Formalized.
- Looking Ahead.
- A Grammar for Expressions.
- Disambiguating the Expression Grammar.

BNF grammars

- Formally, BNF grammars are four-tuples that contain
  - Sigma, the alphabet from which strings are built (note that this alphabet may have been generated from the original input via a lexer). These are also called the terminal symbols of the grammar.
  - N, the nonterminals in the language. You can think of these as names for the structures in the language or as names for groups of strings.
  - S in N, the start symbol. In BNF grammars, all the valid utterances are of a single type.
  - P, the productions that present rules for generating valid utterances.
- The set of productions is the core part of a grammar. Often, language definitions do not formally specify the other parts because they can be derived from the grammar.
- A production is, in effect, a statement that one sequence of symbols (a string of terminals and nonterminals) can be replaced by another sequence of terminals and nonterminals. You can also view a production as an equivalence: you can represent the left-hand-side by the right-hand-side.
  - If we read productions from left to right, we have a mechanism for building strings in the language.
  - If we read productions from right to left, we have a mechanism for testing language membership.
- What do productions look like? It depends on the notation one uses for nonterminals and terminals, which may depend on the available character set.
- We’ll use words in all caps to indicate terminals and words in mixed-case (or all lower case) to
represent nonterminals, and stuff in quotation marks to indicate particular phrases. We’ll use ::= to separate the two parts of a rule.

- We might indicate the standard form of a Pascal program with

```
Program ::= PROGRAM
           IDENTIFIER
           OPEN_PAREN
           identifier-list
           CLOSE_PAREN
           SEMICOLON
           declaration-list
           compound-statement
           END
           PERIOD
```

- Similarly, we might indicate a typical compound statement in Pascal with

```
compound-statement ::= BEGIN
                      statement-list
                      END
```

- Lists of things are often defined recursively, using multiple definitions. Here’s one for non-empty statement lists. It says, “a statement list can be a statement; it can also be a statement followed by a semicolon and another statement list”.

```
statement-list ::= statement
statement-list ::= statement SEMICOLON statement-list
```

- The statements may then be defined with

```
statement ::= assignment-statement
statement ::= compound-statement
... assignment-statement ::= IDENTIFIER BECOMES expression
...```

- At times, we may find more concise or easy to manipulate ways of writing these rules. For example, in Yacc and Bison you are expected to use each nonterminal once on the LHS but to use an or (|) to join additional options. We terminate the options with a semicolon.

```
statement-list : statement
                : statement SEMICOLON statement-list
```

- How do we use BNF grammars? We show derivations by starting with the start symbol and repeatedly applying productions. Eventually, we end up with a sequence of terminals.
  - Any string that can be produced in this way is in the language.
  - The attempt to show that a string is in a language given by a BNF grammar is called parsing that string.
Applying Rules

- In essence, you use a grammar to show that a string is derivable from a regular expression.
- Consider the grammar for palindromes over A and B.

```
pal : epsilon
    | A pal A
    | B pal B
    | A
    | B
    ;
```

- We know that ABBBA is a palindrome
- We can derive the palindrome by repeatedly expanding nonterminals

```
pal -> A pal A
    -> A B pal B A
    -> A B B B A
```

- We can start with the terminals and work our ways backwards

```
A B B B A > A B pal B A
    > A pal A
    > pal
```

- We will see advantages and disadvantages to each approach.

Looking Ahead

- As you might guess, we’re going to do the same thing for parsing that we did for lexical analysis: We’ll consider how to go from a statement of the syntax of a language to a program that analyzes inputs to see if they match that syntax.
  - That will take a number of days.
- In this case, the translation from specification to implementation may impose some additional restrictions on the rules we can write.
- And, as in the case of f?lex, we may annotate each production with some code to execute when the production is used.

An Expression Grammar

- Expressions occur in most programming languages.
- So let’s write a grammar for infix expressions over variables and integers.
- The primary nonterminal: exp
- “Base cases”

```
exp ::= _INTEGER
exp ::= _IDENTIFIER
```

- Combining two expressions with a binary operator
exp ::= exp binop exp
binop ::= _PLUS
binop ::= _DASH
binop ::= _STAR
binop ::= _SLSH

- Unary operators

exp ::= exp unop exp
unop ::= _PLUS
unop ::= _DASH

- Parenthetical expressions

exp ::= _LPAREN exp _RPAREN

- This grammar is ambiguous: There are multiple parse trees for the same expression.

3 + 4 * 5
a - b - c

- What do we do about it?

Disambiguating Expressions

- So, how do we make the expression grammar non-ambiguous?
  - Actually, let’s do a simpler grammar; let’s eliminate unary operators.
- We reflect a bit on what is and is not legal.
- We are unhappy with a tree with a multiplicative operator at the root that has an unparenthesized additive operator in either subtree, because that binds addition closer than multiplication.
  - It’s okay if the addition is parenthesized, since parentheses have higher precedence.
- We are unhappy with a tree with a multiplicative operator at the root that has an unparenthesized multiplicative operator in the right subtree, because that violates left associativity.
- Similarly, we are unhappy with a tree with an additive operator at the root that has an unparenthesized additive operator in the right subtree, because that violates right associativity.
- So, we create a few new nonterminals to correspond to the different kinds of expressions
  - exp: Anything is fair game. Parenthesized operations. Unparenthesized addition operations. Unparenthesized multiplication operations.
  - term: No unparenthesized addition operations. These correspond to the things that can fall to the right of an addition operation or the left of a multiplicative operation.
  - factor: All operations are parenthesized. These correspond to the things that can fall to the right of a multiplicative operation.
- Let’s start with factors and work our way up.
  - Things can be parenthesized. In that case, everything is parenthesized.
    factor ::= _LPAREN exp _RPAREN
  - Things can have no operation. In that case, there are no unparenthesized operations.
factor ::= _IDENTIFIER
factor ::= _INTEGER

- And that’s about it.
- Now, let’s move on to terms. We said that a multiplicative operator can have an unparenthesized multiplicative operator to the left (term), but not to the right (factor).

term ::= term mulop factor
mulop ::= _STAR
mulop ::= _SLASH

- Is that enough? It suggests that every term must have a multiplicative operation. But you can certainly have expressions that have no unparenthesized addition that have no multiplication.
- No multiplication + no unparenthesized addition =gt; factor

.term ::= factor

- Addition is left as a task for the reader.
- Suppose we wanted to add exponentiation (higher precedence than multiplication). What would we do?
- Suppose we wanted to add comparison operations (lower precedence). What would we do?