CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Lexical and Syntax Analysis

(Follows the Sebesta Text Chapter 4)

Lexical and Syntax Analysis

Languages can be compiled, interpreted, or both (hybrid systems - convert to some intermediate form).

Samples are C, Python, and Java (respectively).

Lexical and Syntax Analysis

All of these languages (and all of those types of languages) need lexical and syntax analyzers.

Lexical Analysis

A **Lexical Analyzer** is basically just a pattern matcher.

Given some arbitrarily large string, it attempts to find substrings matching certain patterns.

The output from the lexer is tokens.

Lexical Analysis

We studied this in the Lexing and Regex lesson.

Syntax Analysis

Grammar syntax analysis begins with the parser. A parser takes the output tokens from the lexer and attempts to create a parse tree.

Syntax Analysis

Parsing can be of two types:

- **Top-Down** - builds the parse tree starting at the root and working down to the leaves.
- **Bottom-Up** - built from the leaves up to the root.

Syntax Analysis

Top-down parsers perform a preorder traversal (meaning they start with the root).

This corresponds to how we worked out derivations before. We began with the start symbol, and performed the left-most derivation at each step.

Syntax Analysis

The most common top-down parsers are recursive-descent parsers (non-backtracking). These is a type of LL algorithm; the first L means we scan from left to right, and the second L means we perform a leftmost derivation for each sentential form.

Syntax Analysis

For instance, with this grammar:

```
<program> -> begin <stmt_list> end
<stmt_list> -> <stmt>
             |  <stmt> ; <stmt_list>
<stmt> -> <var> = <expression>
<var> -> A | B | C
<expression> -> <var> + <var>
              |  <var> - <var>
              |  <var>
```

Syntax Analysis

The sentence `begin B = A + C end` would be derived as such:

```
<program>
begin <statement_list> end
begin <stmt> end
begin <var> = <expression> end
begin B = <expression> end
begin B = <var> + <var> end
begin B = A + <var> end
begin B = A + C end
```

Syntax Analysis

A Bottom-Up parser starts at the leaves and works upward to the root.

This results in a reverse rightmost derivation.

Syntax Analysis

This method would then begin with the statement/sentence and work up to the start token.

Because nonterminals can have more than one production (RHS), this can be tricky.

Syntax Analysis

Finding the correct RHS is complicated and relies upon first finding the "handle" - the correct RHS that allows the parser to continue.

Syntax Analysis

Consider (from the book):

```
S -> aAc
A -> aA | b
```

And the sentence aabc.

```
aabc
aaAc
aAc
S
```

Syntax Analysis

This is easy in this particular instance, as the sentence `aabc` only contained one possible RHS.

Syntax Analysis

Bottom-Up parsers are an LR algorithm. This means that they proceed from left to right in scanning the input and generate a rightmost derivation.

Syntax Analysis

Bottom-Up Parsers tend to be more efficient, and are more common.

Top-Down Parsers may need to perform guesses that require backtracking, or complicated look-ahead mechanisms (slow) to progress.

Syntax Analysis

Recursive Decent Parsers are one type of top-down parser. They are composed of multiple subprograms, many of which are recursive.

There will be a subprogram for each nonterminal in the grammar.

Syntax Analysis

For instance:

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
*/
   void expr() { printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
while (nextToken == ADD_OP || nextToken == SUB_OP) { lex();
  term(); }
  printf("Exit <expr>\n");
}  /* End of function expr */
```

Syntax Analysis

Note that though the subprogram checks the next token it doesn't actually remove it from `nextToken`. It will be consumed in the parsing in `term()`.

This will allow us to check for syntax errors at this stage though. If we are expecting a token and find something else, we error out.

Syntax Analysis

Recursive descent parsers are LL parsers. This means they are plagued by the problems of LL parsers.

One big problem with LL parsers is that they don't handle left recursion well.

Syntax Analysis

For example:

```
A -> A + B
```

This statement will call itself infinitely (recall that we leave the token until the subprogram consumes).

Syntax Analysis

The grammar we just saw contained what we call **direct left recursion** (the recursion happened all in one rule). We may also have **indirect left recursion**:

```
A -> B a A
B -> A b
```

We can see in this code that B will call A and A will call B infinitely.

Syntax Analysis

Left recursion is a problem for all top-down parsers (but not bottom-up!).

Another problem with top down parsers is that they don't always know which RHS to take simply by looking at the next token. Sometimes not only is it hard, but impossible.

We can test whether or not it is possible for a given non-left recursive grammar using the **pairwise disjoint test (PDJ)**.

Syntax Analysis

The PDJ helps us to compute a set from the RHSs of a nonterminal. We call these sets **FIRST** and compute them as follows:

- For each nonterminal (A) in the grammar that has more than one RHS, the intersection of the RHS sets must be empty.

```
A -> aB | bAb | Bb
B -> cB | d
```

Our sets then are `{a}, {b}, {c,d}` (B yields the c,d possibilities). These are disjoint, so we can decide how to proceed.

Syntax Analysis

However!

```
A -> aB | BAb
B -> aB | b
```

Our sets here are `{a}, {a, b}`. These are not disjoint as they have an intersection of `{a}`. This means the parser cannot tell which RHS to choose based upon the next token.