Compiler Construction (CS - 636)

Sadaf Manzoor

Outline

1. Top-Down Parsing
2. Recursive-Descent Parsing
3. Summary

Top-Down Parsing

Lecture: 11 &12

Top-Down Parsing

- A top-down parsing algorithm parses an input string of tokens by tracing out the steps in left-most derivation
  - Parse trees implies preorder tree traversals
- Top-down parsers come in two forms; **backtracking parsers** and **predictive parsers**
  - A predictive parser attempts to predict the next construction in the input string using one or more lookahead tokens
  - A backtracking parser will try different possibilities for a parse of the input, backing up an arbitrary amount in the input if one possibility fails

Top-Down Parsing (Continue…)

- Backtracking parsers are more powerful than predictive parsers
- Backtracking parsers are much slower, requiring exponential time in general and, therefore, are unusable for practical compilers
- Two popular top-down parsing algorithms
  1. Recursive-Descent Parsing
  2. LL(1) Parsing

Recursive Descent Parsing

- The idea of Recursive Descent Algorithm is simple;
  - We view the grammar rule for a nonterminal A as a definition for a procedure that will recognize an A
  - The right hand side of the grammar rule for A specifies the structure of the code for this procedure
  - The sequence of terminals on RHS correspond to matches of input
  - The sequence of non terminals on RHS correspond to call to other procedures
  - The choices on RHS correspond to alternatives (case or if-statements) within the code

Recursive Descent Example

Grammar rule:

factor → (*exp*) | *number*

Code:

```
void factor(void) {
  if(token == number)
    match(number);
  else {
    match('(');
    exp();
    match(')');
  }
}
```

Recursive Descent Example (Discussion)

- How lookahead is not a problem in this example?
  - if the token is *number*, go one way, if the token is '(' go the other, and if the token is neither, declare error

```
void match(Token expect) {
 if (token == expect)
    getToken();
  else
    error(token,expect);
}
```

Error Handling in RD is Tricky!

- If an error occurs, we must somehow gracefully exit possibly many recursive calls
- Best solution: use exception handling to manage **stack unwinding** (which C doesn't have!)

- But there are worse problems: left recursion doesn't work!

Left Recursion is Impossible

- Grammar

  exp → *exp addop term* | *term*

- Code

```
void exp(void) {
  if (token == ??) {
   exp(); // uh, oh!!
   addop();
   term();
  } else
   term();
}
```

Repetition & Choice: Using EBNF

- Consider the following grammar and try to write its pseudo code for recursive descent;

  if-stmt → **if (** *exp* **)** *statement* |
  if (*exp* **)** *statement* **else** *statement*

- Consider the following grammar and try to convert it in recursive descent compatible grammar:

  exp → *exp addop term* | *term*

Summary

Any Questions?