Writing a Parser

COMP360

“Without the fun, none of us would go on!”

Ivan Sutherland

“father of computer graphics”

Parsing Methods

- Top Down Parsing
  - Traces a leftmost derivation
  - Builds a parse tree in preorder
  - LL and Recursive Descent are top down
- Bottom Up Parsing
  - Builds a parse tree starting at the leaves
  - Produces the reverse of a rightmost derivation
  - LR and LR-K algorithms are bottom up

Recursive Descent

- Relatively easy algorithm to implement
- Functions or methods are written to represent each BNF production

List of Tokens

- The lexical scanner creates a list of tokens
- The tokens are best implemented as an array, ArrayList or linked list of Token objects
- The tokens include the value of the token, the type and their location in the source code

Input function

- There is an input function that gets the next token from the lexical scan

```
static Token getNextToken() {
        if (curTok >= tokenList.size()) {
            return null;                    // no more tokens
        }
        return tokenList.get(curTok++);
    }
```

- I also create a method, peek(), that returns the next token, but does not remove it from the input queue

equation → varconst | varconst equation

```
boolean equation() {
   Token nextVar = getNextToken();       // get next variable
    if (!nextVar.isConstVar()) {
        return false;
    }
    while (peek().isConstVar()) {         // more variables?
        getNextToken();                   // get the next variable
     }
     return true;
}
```

True, False or Worse

- The recursive descent methods can be boolean returning true if the BNF production matches and false if it does not
- My programs throw an exception if a syntax error is found

Example Grammar

expr → term | term **+** expr

| term **-** expr

term → factor | factor * factor

| factor **/** factor

factor → **variable** | **(** expr **)**

expr function

expr → term | term + expr
| term - expr

```
boolean expr() {
  term();
  while (peek() == '+'
         || peek() == '-') {
     getNextToken();
     expr();
  }
  return true;
}
```

Context Free Grammar

- Context Free Grammars can be recognized by a Finite State Automata with a stack

- Where is the stack in the recursive descent algorithm?

Left Recursion

- Left Recursion occurs when a production defining a non-terminal starts with the non-terminal

  thing → thing + widget

- Indirect Left Recursion has the same effect as direct recursion, but involves multiple rules

  thing → dodad + widget

  dodad → thing x

Recursion Problems

- The recursive descent algorithm will get stuck on with a left recursive rule

```
void thing() {
   thing();          // infinite loop here
   if (nextToken != "+") Syntax error
   widget();
}
```

Right Recursion Correction

- The problems with left recursion can be corrected with right recursion.

thing → thing + widget

- can be changed to

thing → widget + thing

Left Recursion Example

exp → exp **+** term
| term

term → term ***** factor
| factor

factor → **(** exp **)**
| varnum

Corrected BNF

exp → term **+** exp
| term

term → factor * term
| factor

factor → **(** exp **)**
| varnum

Unique First Terminal

- It is difficult to parse productions that have multiple options that start with the same symbol

```
thing → x widget
      | x doodad
```

- List recursion is generally not a problem

```
A → B | B , A
```

Unique Symbol Grammar

- The previous example can be corrected by adding another non-terminal

thing → x whatever

whatever → widget | doodad

Consuming Tokens

- If you have a production like

  `whatever → widget | doodad`

- The widget method will consume the tokens checking if the input is a widget
- Calls to getNextToken will advance the current token pointer
- When the whatever method calls doodad, it will be looking past the tokens checked for widget

Backing Up

- If a method fails, it should reset the current token pointer to where it was when it started
- You can save the current token pointer when the method starts and reset it if the method returns false

How Big is the Program

- My lexical scanner is 150 lines
- My Token class is 125 lines
- My parser for Snowflake is 375 lines

- My program has lots of comments, which make them longer