

---

# CS 421 – Spring 2007

## Lecture Notes Set 22:

### Introduction to Recursive Decent Parsing

Elsa L. Gunter<sup>1</sup>  
Transcribed by: Pooja Mathur

---

#### Corresponding to Slides:

Made available: March 14, 2007

Revision 1.0

---

## 1 Change Log

1.0 Initial Release.

## 2 Writing parsing programs

This information is an introduction of what we will discuss after break, it will not be on the exam.

### 2.1 Programs for parsing

So far we have talked about grammars as a way of describing languages and recognizing strings in a way to head us towards semantics using parse trees. We've done this in a very mathematical fashion where you look at it and see how it's broken up and are able to perform the parsing with knowledge of the entire string as a whole. This is not useful when it comes to having the computer do this. So this means we need a more systematic way of being able to tell if a string is accepted by a grammar. Also, we don't want the program to tell us if the syntax works, we want it to actually do something with the strings we send it.

So what we want is a program that can produce elements of a datatype that represent the parse trees for expressions. This process is broken up into a couple pieces. There is the lexer that generates the tokens from a string or character stream. It tells if the string/stream satisfies being part of a syntactic category. That is, it figures out if the string/stream represents an identifier or a keyword, float, integer, etc. The lexer analyzes the string and bundles them into units called tokens. Then the lexer is called by the parser that generates the parse tree from the token list or stream. The parser takes the tokens list and treats them like letters of an alphabet, and attempts to build a parse tree.

### 2.2 Idea of Recursive Decent Parsing

One way to write a parser is by recursive decent parsing. Like its name suggests, this is going to be done by writing a collection of recursive functions. The recursive functions are going to parallel the rules of the grammar, so that each time we get to a new non-terminal, it is going to correspond to a new recursive call. It is going to build a parse tree, moving from left to right, and it is going to build the tree in a top down fashion.

So, each non-terminal is going to correspond to a subprogram in the overall parser. This subprogram will parse any token that this certain non-terminal can generate. The right side of the rules correspond to the recursive call for the next part of the token stream to be parsed. Now, since we are looking at the stream from left to right, the subprograms must be able to decide how to parse by looking at the left-most character of a string. This puts a restriction on our grammar. Since we have to move from left to right...that is left will be near the top of the tree, recursive decent parsers

---

<sup>1</sup>© 2007, Share and Enjoy

cannot be made from left-recursive grammars. The recursive part has to be on the right side of the rule to make this work. We will learn later of a procedure to parse these left-recursive grammars.

Now, having to decide right then and there, what rule you are going to use might be awkward to say the least. Look at this grammar as an example:

$$\begin{aligned} \langle \text{expr} \rangle &::= \langle \text{term} \rangle \mid \langle \text{term} \rangle + \langle \text{expr} \rangle \mid \langle \text{term} \rangle - \langle \text{expr} \rangle \\ \langle \text{term} \rangle &::= \langle \text{factor} \rangle \mid \langle \text{factor} \rangle * \langle \text{term} \rangle \mid \langle \text{factor} \rangle / \langle \text{term} \rangle \\ \langle \text{factor} \rangle &::= \langle \text{id} \rangle \mid ( \langle \text{expr} \rangle ) \end{aligned}$$

Look at the first rule, the parser has to be able to tell, when it sees an  $\langle \text{expr} \rangle$ , if it is going to break it up into a  $\langle \text{term} \rangle$  or  $\langle \text{term} \rangle + \langle \text{expr} \rangle$  or  $\langle \text{term} \rangle - \langle \text{expr} \rangle$ . All it has to go on is that the first character is a  $\langle \text{term} \rangle$ . But each of the three rules begins with  $\langle \text{term} \rangle$ , so what is it going to do?  $\langle \text{term} \rangle$  has the same problem with its rules. They all begin with  $\langle \text{factor} \rangle$ . We will learn how to deal with this after spring break.