CSC173: Project 2

Recursive Descent and Table-Driven Parsing

The goal of this project is to demonstrate your understanding of the formal model of context-free grammars and parsing by applying the principles of the model to a specific grammar (details below).

The goal of the project is not simply for you to write a program that parses strings from the specific required grammar. The goal is for you to demonstrate how the formal model allows you build a parser based on the grammar mechanically with almost no thinking required.

Process not product.

Given a parsable grammar, for this project you must first construct a recursive-descent parser for the grammar using parsing functions, as seen in class and in the textbook. These parsing functions come almost directly ("mechanically") from the productions of the (parsable) grammar, as seen in class. You will demonstrate your parser by asking the user for input strings, parsing them, and printing the resulting parse tree (or an error message).

You must also implement a table-driven parser for the grammar, following the description seen in class and in the textbook, and demonstrate its use. Again: it's not about being a creative programmer. It's about understanding and applying the formal model. In this case, your code should be able to parse strings using any grammar given a parsing

table connected to the productions of the grammar. But you will only need to show it working on one grammar.

Please read the requirements carefully.

Language

For this term, the language that we will be parsing is a very simple programming language.

Here is an informal description of the language:

• An program is zero or more statements, each followed by a semicolon.

• There are only two types of statements:

– Assignment (“=”): assign the value of an expression to a variable

– Output (“>”): print the value of an expression on the console

• Variables are named with a single letter (“a” to “z”).

• Expressions use unsigned integers, the four arithmetic operators, and parentheses, as seen in class and in the textbook.

– The expression “<” (less than) reads and returns a number from the console.

• Whitespace (spaces, tabs, newlines) is not allowed. The entire input must be on one line, without a newline at the end.1

Here are some well-formed expressions of this language (one per line):

```
x=1;
>123+1+2/2*3;
a=<;b=<;>(a+b)/2;
```

The last of these is actually a useful program. You should be able to figure out what it

does. If not, please go to study session ASAP.

In order to build a parser, we need a context-free grammar for the language and in order to use the required parsing methods (seen in class and in the textbook), that grammar must be parsable by a recursive descent or table-driven parser.

Think about this yourself.

After you have thought about it yourself, please read Appendix A. Note that you MUST use the grammar given in this document.

I know that this makes the strings hard to read and would not be good for a real programming language. However allowing it without making the grammar impossibly complicated involves using a lexical

analyzer (a.k.a. scanner or tokenizer). This is mentioned in a footnote on p. 617 of the textbook, and you

will see it if you go on to take CSC254.

Requirements

Part 1: Recursive-descent parser (60%)

Implement a recursive-descent parser that produces parse trees for expressions of the required language.

• You MUST use the grammar given in this document.

• The style of the parsing functions MUST be as seen in class (which is sort of like what is in the textbook; check out the code available on BlackBoard with the project description).

• Your parsing functions MUST use functions lookahead and match as seen in class.

• You should be able to create the parsing functions by reading the productions of the grammar with almost no thinking required.

• The only place where thinking is required is how to use the lookahead symbol, as seen in class and in the textbook.

• This will be boring if you do it right. I'm sorry.

You must then demonstrate your parser by reading strings from the user, parsing them, and printing the resulting parse tree.

• Your program must prompt the user for input.

• You may assume that expressions are on a single line of input, and no longer than 255 characters. Note that the empty string is a valid input, so you should use fgets rather than scanf to read a line of input. See the "C for Java Programmers" guide and/or ask in study session if you need help with this.

• If the input is not well-formed, your parser should print an error message and resume processing at the next line of input.

• Otherwise print the parse tree as described below.

• This phase of the program should terminate when it reads the string "quit" (without the quotes).

Part 2: Table-driven parser (40%)

Implement a table-driven parser for expressions of the required language.

• As for Part 1: read expressions from the user, try to parse the input, print the parse tree or an error message, until "quit".

• Most of the infrastructure of the parser will be the same as for Part 1.

• You MUST use the grammar given in this document.

• You MUST use an explicit parsing table that references explicitly-represented productions (FOCS Figs. 11.31 and 11.32). That means the table must contain either

indexes of productions in a list or array of productions (as seen in the textbook), or

references (pointers) to the productions themselves.

• You MUST have a function that creates and returns an instance of this table for

the grammar of the required language. This function does not need to translate

a grammar into a parsing table. It just needs to build and return the parsing table

for the grammar that you are using. So work it out by hand, then write the code to

produce it.

• You MUST have a parsing function that takes a parsing table and an input string as

arguments and does a table-driven parse. (You may have helper functions also.)

Note that this function should be able to parse using any grammar, given its parsing

table and productions. That is, it is independent of the grammar that it is using

(but you only need to show it working with the required grammar).

• It may be helpful to produce output like FOCS Fig. 11.34 during debugging.

The next step would be to convert the parse trees produced by your parser(s) to expression trees and then either evaulate the expressions to compute their values or generate

code to compute the values later. It's not hard to do that once you've built the parse

trees, but you do not have to do it for this project. You will see it in CSC254 if you take

that course.

Your project MUST be a single program. This program should read the input and then

call each parser (assuming you do both parts) and print the results, as described above.

It is your responsibility to make it clear to us what your program is doing.

There is no opportunity for extra credit in this project.

Figure 1: Example parse tree and corresponding printed output

Parse Trees and Printing Parse Trees

A parse tree is a dynamic data structure. You have seen trees in Java and they're the same in C. The textbook has an entire chapter on trees (Chapter 5), and the chapter for this unit has useful code also (also available on BlackBoard with the project description).

For this project, your program must print the resulting parse tree to standard output.

There are many ways to do this, but for this project you will produce output in an indented, pretty-printed format. What this means is:

• Each node is printed on a separate line.

• The children of a node are indented relative to their parent.

• All children of a node are at the same level of indentation.

• The empty string symbol epsilon "ε" can be printed in C using by using a Unicode escape: "\u03B5". If that doesn't work for you, just print the word "empty".

Printing a tree involves doing a tree traversal, right? Traversing a tree is a recursive procedure, right? You print nodes and you recursively print their children, in the right order. So you can implement it using a recursive function. It's elegant and practical.

You also need to keep track of the current indentation level. So this will be a parameter to your pretty-printing function. In C, which does not have function overloading, this usually means two functions: a toplevel pretty-print function with no indentation parameter, and a helper function with that parameter, called from the toplevel function with indentation

0 to get the ball rolling.

If this did not make sense to you, please get to study session ASAP.