3predictive-parsing-1-lab

CST 334 (Operating Systems)
Dr. Glenn Bruns

Lab: Predictive parsing 1

In this lab we'll write a BNF grammar, then write a predictive parser for the grammar. Refer to the hints at the end of this document only when stuck!

Here is a BNF grammar for a comma-separated list of one or more variables:

vars ::= vars1 | ""
vars1 ::= vars1 "," **var** | **var**

1. Is this grammar left-recursive?

2. Without looking at the lecture slides, can you transform the grammar to remove left-recursion?

Let's look at a simple predictive parser.

3. On mlc104, create a directory for this lab, then copy `/home/CLASSES/brunsglenn/cst334/labs/pred-parser/pred-parser.tar` to the directory you created. Untar the file (you remember how right? hint: e**x**tract).

4. Look at file parser.c. It is a predictive parser. In the comment about function nums(), you can see the BNF the parser is based on. Read the BNF carefully, and look at the corresponding code.

5. Look at file input.txt. It is some digits, each followed by a semicolon.

6. Look at the main program.

7. Look at the Makefile. What will happen when you run 'make'?

8. Run make. You should see the code compiled, the parser run on input.txt, and 'parsed' as output. If not, something is wrong.

Now we're ready to write our own parsing functions.

9. Create a file grammar.txt. Write your BNF for step 2 in the file.

10. In parser.c, create a predictive parser for non-terminals vars by writing a function vars() based on the BNF you just put in grammar.txt. You will want to create one or more additional functions if your BNF has multiple non-terminals. Remember, the lexer returns ID when it sees an identifier (see lexer.h).

11. Modify the function parse() in parser.c so that it calls your function vars() instead of nums(). Near the top of the file, change 'void nums()' to 'void nums(), vars()'. Add any other functions you created to this list.

12. Edit file input.txt so that it is a comma-separated list of variable names (e.g. 'x,y,z').

13. Run make to test your code. Also, run at least two more tests by changing input.txt and running again. Also, try input that should not be valid, such as "x,y,".

Now we'll make further extensions. Perform the above steps 9-13 above for each of the following extensions:

14. Extend your grammar to allow print statements like these:

print()

print(x)

print(x,y,z)

If you look in lexer.h, you'll see that PRINT is a constant that the lexical analyzer knows about. So you can write things like match(PRINT).

If you still have time, work on the following:

15. Extend your print statement so that it allows a comma-separated list of expressions to be used, where an expression is either a variable name, or has the form $NUM.

16. If you still have time, allow for a "program" of the form { print statement }. In other words, a print statement between curly braces. It's a simple form of awk program.

17. If you still have time, allow for multiple print statements between curly braces. Require that each print statement end with a semicolon.

18. If you still have time, allow for a program to consist of multiple blocks, where each block is curly braces that enclose a bunch of print statements.

Hints

1. Yes, because one of the productions for non-terminal 'vars1' begins with 'vars1'.

2. Using the rule we learned in class, an equivalent grammar is:

vars ::= vars1 | ""
vars1 ::= **var** vars2
vars2 ::= "," **var** vars2 | ""

Note that by replacing vars1 by its definition, we get this equivalent grammar:

vars ::= **var** vars2 | ""
vars2 ::= "," **var** vars2 | ""

3. -

4. -

5. -

6. -

7. -

8. -

9. This is very similar to what was done in lecture. You probably want two non-terminals

10. Use two functions, one for each non-terminal. They will be similar to function nums(). When you want to match on a character, like a comma, you can write `match(',')`.

11. -

12. -

13. -

14. This is easy: there will be only one production.

15. Don't require that $ and the NUM be right next to each other. Spaces between them are okay. I suggest creating functions exprs(), which takes care of a list of expressions, and expr(), which can be either an ID or $ followed by NUM.

16. -

17. -
18. -