CST 334: Operating Systems

Dr. Glenn Bruns

# Building a predictive parser

**Purpose.** The purpose of this assignment is to help you to bring together everything you've learning about syntax and parsing.

**Instructions.** This file on mlc104:

/home/CLASSES/brunsglenn/cst334/hw/hw13/prog.tar

contains starter code for a parser. The parser is for a simple language. In this language:

* a programis one or more statements, and each statement must end with a ";"
* a statement is either 1) a function call with exactly one argument, or 2) an assignment statement. The argument to the function call is an expression.
* an expression is either 1) a function call with exactly one argument, 2) a variable, or 3) a non-negative integer

Here's an example program that shows most of the features of the language:

y = 3;

z = y;

x = abs(z);

print(x);

Note that function calls always contain exactly one expression within parentheses. 'abs' and 'print' are just two example function names -- other function names are allowed.

Here is a BNF grammar for the language: (prog is the start symbol)

prog ::= ID stmt1 ; prog1

prog1 ::= ID stmt1 ; prog1 | ""

stmt1 ::= ( expr ) | = expr

expr ::= ID ( expr ) | ID | num

I have written the grammar in a form where it is almost ready to be used for predictive parsing.

Your job is to complete the code in prog.tar so that it will parse a program in this language, and then print it. The file prog.tar contains not just code, but a Makefile and some test files you can run.

Here are how the C files in prog.tar fit together:

* parser.c contains the parsing code. You will be extending this file so that it contains a function for each non-terminal in your BNF grammar. For example, if your grammar contains a non-terminal 'expr', then you will add a function expr() to parser.c
* prog.c, stmt.c, and expr.c are used to build a syntax tree for a program. The function prog() in parser.c has some example code that shows how they can be used to build a parse tree. You will be modifying these files by implementing the print functions in them. For example, in prog.c you will fill in code for the prog\_print() function.
* you don't need to modify any of the other C files

Please do the following:

* Copy the tar file to your own directory, untar it, and understand it. Start with main.c.
* Compile the code (see the Makefile). It should compile without errors.
* Write a BNF grammar for the language. You may need to transform the grammar so it is in the form needed for a predictive parser.
* Edit file parser.c to create a predictive parser based on your BNF (see comment "YOUR CODE HERE")
* Edit your parser so that it returns a 'prog' object, which in turn contains 'stmt' and 'expr' objects
* Edit files prog.c, stmt.c, and expr.c to implement the print functions in those files (see comments "YOUR CODE HERE"

Do NOT edit any other files, and do not edit the files above except as explained in the comments. When I test your code I will only use your parser.c, prog.c, stmt.c, and expr.c files.

The main program will call your parser to get a 'prog' object, then print the 'prog' object to reconstruct the program from its syntax tree.

**Testing your code**. Tests files test1.sh, test2.sh, etc. are in prog.tar and can be used to test your code. When I grade your code I will use these kinds of tests.

You might also like to use program run-tests.sh to test your code. This program runs your parser on all the .p (program) files and displays the result. Here's what happens when I use run-tests.sh with my parser code.

./run-tests.sh

------ test1.p ------

y = 3;

print(y);

------ output: ------

y = 3;

print(y);

------ test2.p ------

y = 3;

x=y;

print( x);

z=sign(x);

------ output: ------

y = 3;

x = y;

print(x);

z = sign(x);

------ test3.p ------

y = 3

x = y;

------ output: ------

line 2: syntax error

------ test4.p ------

y = 3; print(y);

------ output: ------

y = 3;

print(y);

------ test5.p ------

------ output: ------

line 3: syntax error

------ test6.p ------

y = 3;

x = abs(y);

print(x);

------ output: ------

y = 3;

x = abs(y);

print(x);

------ test7.p ------

y = 3;

x = abs(y);

print;

------ output: ------

line 3: syntax error

A handy thing about run-tests.sh is that if you write your own .p files to test your parser, run-tests.sh will use them.

Look at the output above from test programs test2.p and test4.p. The output is formatted differently from the original program because the output was derived from a syntax tree, not the original program text.

**Hints for developing your BNF**. You need to modify the BNF grammar above such that, for every non-terminal in your grammar, either: 1) there is only a single production for that non-terminal, or 2) each production starts with a token (possibly "") and each of these productions start with a different token. Don't forget the techniques we learned in class of modifying a grammar.

**Hints for developing your parser**. We are using a predictive parser, so there will be a function in your parser for every non-terminal in your BNF grammar. Also -- and this is an important hint -- you might find that it helps to have some of the functions in parser.c take arguments. For example, one or more of the functions that correspond to non-terminals in your grammar might take arguments.

**Hints for developing/debugging your code**.The lexer\_num\_val() and lexer\_id\_val() give the integer or string associated with the current lookahead value. If you call lexan() or match(), you'll change the current lookahead value.

**Other hints**. The code in this assignment differs from the code of the previous assignment in some minor ways. 1) In the previous assignment the lexical analyzer returned a string; in this assignment it returns a number. 2) In the previous assignment each function corresponding to a non-terminal began by getting the lookahead token; in this assignment each such function assumes the next token is already stored in variable 'lookahead'.

This code is very similar to the "calculator" code we covered in class. You can find this code on mlc104 in directory

/home/CLASSES/brunsglenn/cst334/labs/calc-parser

**Submitting.** Submit **parser.c**, **prog.c**, **stmt.c**, and **expr.c** as four separate files on iLearn. Do not submit a tar file, gzip file, or other archive file.

**Grading**. I will run 7 tests, including some of the tests above. Some will be code with syntax errors, and some will be code with no syntax errors. Each test is worth 10 points for a maximum total of 70. No points will be awarded if your parser is not a predictive parser! There must be a function for every non-terminal in your grammar.