CSC254/454 Assignment 3: Parser

University of Rochester

September 17 2018

1 Introduction

This assignment will build on the knowledge you gained in the last assignment. Given your prior experience the grammar you will be given this time will be significantly more challenging. In this assignment, you will be given an ambiguous grammar that describes a subset of the C language, and build an LL(1) grammar. Using that new grammar, you will build a parser that will check if an input program is syntactically valid, and will output additional information if it is.
As in the previous assignment you may **NOT** use lex, yacc, or regular expressions, or any kind of special library or generator. You may use recursive decent parsing or table-driven parsing, though we recommend the former.

2 Requirements of the Assignment

1. As in the tokenizer project, your parser should ignore (but copy) "meta-statements".
2. Convert the given grammar into an unambiguous LL(1) grammar and submit it as part of your *README*.
3. Use look-ahead top-down parsing.
4. For each input program, your parser should report "Pass" or "Error" as the result of grammar analysis.
5. Count the number of global and local variables (*data_decls*), functions (*func_list* but not function declarations), and statements (*statements*) in correct programs (Note: *int a,b;* increases the counter of variables by 2, not 1.).

Output to **stdout** the result in the following format:
Pass
Variables: *num_variables*
Functions: *num_functions*
Statements *num_statements*

OR

Error

For example, for a correct program with 4 variables, 2 functions and 6 statements, your compiler should output at least the following:
Pass
Variables: 4
Functions: 2
Statements: 6
And if the input program fails to pass the grammar analysis, output:
Error

3 The Grammar

For this assignment, we are extending the previous language. It is still a subset of the C language, but it is larger.

There are only integer variables and arrays and no external library calls except for I/O. There are no pointers except for reading the input. In this project, we are concerned with the token definition of the language. Note that tokens are separated either by white spaces (including tabs and line returns) or by symbols such as operators and semicolons. **You will be using the same token language as in Assignment 2**.

3.1 The Grammar Specification

program → data_decls func_list
func_list → ε | func func_list
func → func_decl **;** | func_decl **{** data_decls statements **}**
func_decl → type_name **identifier** (parameter_list)
type_name → *int* | *void*
parameter_list → ε | *void* | non_empty_list
non_empty_list → type_name **identifier** | non_empty_list **,** type_name **identifier**
data_decls → ε | type_name id_list **;** data_decls
id_list → **identifier** | id_list comma **identifier**
identifier → **identifier** | **identifier** [expression]
block_statements → **{** statements **}**
statements → ε | statement statements
statement → assignment | general_func_call | print_func_call | scanf_func_call | if_statement | while_statement
| return_statement | break_statement | continue_statement
assignment → **identifier** = expression **;**
general_func_call → **identifier** (expr_list) **;**
printf_func_call → **printf** (STRING) **;** | **printf** (string **,** expression) **;**
scanf_func_call → **scanf** (string **,** **&**expression) **;**
expr_list → ε | non_empty_expr_list
non_empty_expr_list → expression | non_empty_expr_list **,** expression
if_statement → **if** (condition_expression) block_statements | **if** (condition_expression) block_statements **else** block_statements
condition_expression → condition | condition condition_op condition
condition_op → **&&** | **||**
condition → expression comparison_op expression
comparison_op → **==** | **!=** | **>** | **>=** | **<** | **<=**
while_statement → **while** (condition_expression) block_statements
return_statement → **return** expression **;** | **return;**
break_statement → **break;**
continue_statement → **continue;**
expression → term | expression addop term
addop → **+** | **-**
term → factor | term mulop factor
mulop → ***** | **/**
factor → **identifier** | **identifier** [expression] | **identifier** (expr_list) | **number** | **- number** | (expression)

And of course, the following which your previous tokenizer should already do:
number → any consecutive sequence of digits (0-9) that is not part of an identifier.
string → any string between (and including) the closest pair of quotation marks.
meta_statement → any string begins with '#' or '//' and ends with the end of line ('\n').

Note: letters and digits should be part of the same token unless they are separated by a while space character or a symbol.

4 Parser Design

For this project you must use Ruby or Python (Ruby is preferred). Note that your grade will be based on correctness. Although you may organize your code however you like, your parser should be executable on cycle1 as such: "./parser.{rb, py} input_file.c". Remember that learning these steps is essential to succeeding in future assignments. Please use good software design and organization so that you can refer to this assignment again in the future.

You may reuse the same code structure as the previous assignment (as we'd mentioned). If you have not been able to complete the previous assignment or design it properly, feel free to refer to the solution we have posted and build on it.

Please note that to properly turn the grammar into LL(1), you are expected to use FIRST, FOLLOW, and PREDICT sets as described in Section 2.3 of the book.

5 Division of Labor and Writeup

This assignment should be done **alone** meaning you may not pair up for this assignment. Every student in the class is required to submit his/her own source code and a README (See Below). Write a brief README in no more than **two** pages (plain text is preferred). In this README provide a brief but comprehensive explanation of how your program works. If your program does not work, explain what you attempted, and what you think needed to be done in order to get a working project. Be sure your README describes any features of your code that the TAs might not immediately notice.

6 Turn In and Due Date

On a csug machine, put your write-up in a **README.txt** (*no PDFs*) file in the same directory as your code, and (while still in that directory) run the script **~cs254/bin/TURN_IN** to turn in. The script will package the contents of the directory (and any subdirectories) into a bundle and send it to the TAs for grading (so clean up any mess you might have in the directory first).

The due date for this assignment is: **Sunday September 23, at 11:59pm; no extensions.**

7 Extra-Credit

Having built a parser, you now realize that once you detect a syntax error, you can't really go ahead, and you need to stop parsing.
Have you ever noticed how sometimes the compiler says it expected a parenthesis on one line, and reports another error a few lines below in the *same* compilation? This is called *syntax error recovery* and is discussed in Section 2.3.5 which is not covered in this class. The Companion Site has even more details on the subject. For up to **20% Extra Credit**, implement a *syntax error recovery* mechanism, i.e. instead of just reporting "Error", report what the error is. Of course, you will be expected to follow the proper procedure and mechanism, and not just hard-code error messages here and there! Come up with some creative or thorough test cases, you will be graded on those too!