SIENA*college*
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2023

Problem Set 4: Parser for Little C

Due: 11:59 PM, Monday, October 23, 2023

For this assignment, you will be implementing a parser for a further variant of the *lC* (little C) language you used in the previous assignment. You will use the tokenizer you have developed as the first stage in this larger program that will perform a full syntax analysis of (*i.e.*, parse) a *lC* program.

A parser is a complex program. As such, you are strongly encouraged to form groups of 2 or 3 again for this assignment. You need not maintain the same groups you had for the tokenizer unless you wish to do so.

You can find and run the executable for my solution code for this program on `noreaster.teresco.org` in `/home/cs340/probsets/parser/` .

Commit and push often, and use meaningful commit messages. It's only a small part of the grade, but it's The Right Thing To Do.

Getting Set Up

In Canvas, you will find a link to follow to set up your GitHub repository, which will be named `parser-probset-yourgitname`, for this problem set. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 11:20 AM, Thursday, Ocrober 12, 2023. This applies to those who choose to work alone as well!

Parser Requirements

You will be using the grammar specified on the Tokenizer Problem Set description.

Your tasks are

1. Determine from the BNF grammar which rules need to "make choices" and how they will make those choices. That is, those rules that have two or more options on their right hand side, how will your parser know which rule to apply. For this, you will need to determine what the "first" token set is for each choice.

   For example, the `<iterative-statement>` rule can be either a `<while-statement>` or a `<for-statement>` We can readily determine which of these to apply. If the next token is the `while` keyword, we have encountered a `while` statement, a `for` keyword indicates a `for` statement, and any other token means there is an error.

2. Write a C program `parser.c` that takes as its input a single command-line parameter, the name of a file that contains a *lC* program. It should follow the model of the improved `sebesta-recdescent` example in how it initializes the lexer, and calls the function for the start nonterminal (in this case, `program`).

To get started, combine your *lC* tokenizer code with the basic framework found in the `sebesta-redescent` example. You will need to replace the functions `expr()`, `term()`, and `factor()` with many other functions: one for each of the nonterminals in the *lC* grammar. You should name the functions the same as the nonterminals, but replace dashes with underscores. Some of these functions should be quite short, others have more work to do. In most cases, it will be clear what you need to do from looking at the BNF rule and the "first" tokens that will cause a particular rule to be applied. The trickiest rule might be the `<statement-list>`, which consists of a statement, possibly followed by another `<statement-list>`. In this case, you need to look at the BNF rule that produces a `<statement-list>` to determine when we need to call it again, and when the `<statement-list>` should end.

The output of your program should be primarily through the provided `match`, `entryMsg` and `exitMsg` functions. Any time a function in your parser has determined that part of a rule "matches" a token on the input, call `match` with the current function name (*i.e.*, the name of the BNF rule currently being applied), and an appropriately indented message about the token matched will be printed. This, combined with calls to `entryMsg` and `exitMsg` will result in the "parse tree"-like format of the output.

When you encounter a parse error, call the `error` function with an appropriate message. The messages in my version are short and probably not that helpful in many circumstances. If you get the parser working and still have time, see if you can improve on these messages.

A slow and steady approach will be essential here. You will definitely need to ask questions. You will definitely need to discuss your approach with your partner(s). No one piece is huge, though, so tackle it one step at a time and keep making progress.

General Requirements

Your code should be commented appropriately throughout. Please also include a longer comment at the top of your program describing your implementation. And, of course, it should include your name(s).

Your program should compile without warnings using `gcc` on noreaster when the `-Wall` flag is included. This flag turns on extra warnings that will help you avoid some of the pitfalls of C programming. If you encounter any warnings that you don't know how to fix, ask!

Include a `Makefile` that compiles the program with the `-Wall` flag. This `Makefile` should produce an executable program called `parser`. My `Makefile` is on `noreaster.teresco-.org` in `/home/cs340/probsets/parser/` . Please feel free to use or modify as you see fit.

Submission

Commit and push!

Grading

This assignment will be graded out of 150 points.

Feature	Value	Score
Basic recursive descent parser organization	25	
Parser completeness and correctness	85	
Appropriate output format	10	
Git commit frequency and message quality	10	
Program documentation	15	
Program efficiency, style, and elegance	5	
Total	150	