A Scanner and Parser for DEMO

Programming Assignment 1
COMP 506, Spring 2020

Project Due Date: March 13, 2020 @ 11:59 PM

1 Introduction

This project will give you experience building a scanner and parser for a programming language called DEMO. You will use the `flex` scanner generator and the `bison` parser generator that are distributed as part of most Linux systems — these are standard tools in widespread use throughout the world. The educational goal for the project is to provide you with (1) experience manipulating regular expressions and context-free grammars, and (2) direct experience with activities in three time frames: design time, build time, and compile time.

From a practical perspective, you will need to build a `flex`-based scanner for the microsyntax, or lexical elements, of DEMO. Once you have a scanner, you will need to build a `bison`-based parser that uses the output of the scanner to recognize, or parse, DEMO programs. Your second programming assignment will build on your own code for this assignment.

ADVICE ⇒ All documents for this lab are available from the course web site, as are various reference materials for `flex` and `bison`. Before you begin coding, you should read both the DEMO language definition and this document in their entirety.

2 Project Summary

Your task is to:

1. use the standard, open-source `flex` scanner generator to build a scanner for DEMO;
2. use the standard, open-source `bison` parser generator to build a parser for DEMO, as tokenized by your `flex` scanner.
   We will test the correctness of your scanner and parser by running them on a collection of input DEMOprograms, some of which are provided.

In this project, the output of your parser is either (1) a message stating that the input was parsed and found to be correct, or (2) a set of one or more error messages detailing the syntax errors that your parser discovered and the line numbers where the errors were found.

Your work product for this project is a working scanner / parser for DEMO, written in the C programming language. In the second project, you will extend your scanner / parser to generate ILOC code for the input DEMO program. You should take care in this project to create a code base on which you can build the later project.

3 Receommended Approach

1. Find and read all the documents that are relevant to the project. That should include, at a minimum, this document, the DEMO language specification, materials on `flex` and `bison`.

2. Make sure that your CLEAR account has access to a working C compiler and copies of `flex` and `bison`. The default versions on CLEAR appear to be:

`gcc`	version 4.8.5	use `gcc --version` to verify
`clang`	version 3.4.2	use `clang --version` to verify
`flex`	version 2.5.37	use `flex -V` to verify
`bison`	version 3.0.4	use `bison -V` to verify

   You should be able to log into CLEAR with your Rice NetID. If you cannot, or you cannot see these tools once you are on CLEAR, contact the IT help desk for assistance by sending an email to `help@rice.edu`.

3. Create a `flex` input specification for DEMO, based on the DEMO language specification. Write a small test program that repeatedly invokes your scanner, until it returns an end-of-file indication. Have the test program print out each lexeme and run the test program over the DEMO test programs. In the `flex` manual, an example which creates such a program for a small Pascal-like language is given at the end of Section 4. It does leave out one important detail: along with the main() routine, you need to define a routine yywrap() which simply returns 1. This indicates that you are inputting one .l file to `flex`.

4. Create a `bison` input specification for DEMO, based on the DEMO language specification. You will need to transform the grammar, which includes rewriting it in `bison` input format and eliminating ambiguity. Use `bison` as a tool for eliminating conflicts from your grammar, it issues excellent diagnostics when reportng conflicts.

5. Build the interface between your scanner and your parser. In the declaration section of a `bison` input file, token type names are declared and serve as terminal symbols in the grammar. From this `bison` generates `tokens.h`. You need to add `#include tokens.h` in the definitions section of the .l file for the DEMO scanner.

6. Extend your working parser by improving its error detection and error message capabilities. Insert the `error` token into your grammar in places that will allow the parser to recover and continue—that is, to recognize more than one error in an input file.

   In class we will discuss how to make effective use of the error token and the `bison` macro `yyclearin`. In some cases, you may want to add rules that match a specific error so that you can generate a more precise error message. The test programs named `error*.demo` in c̃omp506/students/demo/lab1/errors are a start. The reference parser has some error detection and reporting built into it.

7. We have created a driver that parses command line arguments and invokes your parser. See the README file on clear at comp506/students/driver for more detail.

4 Submitting Your Work

Name your .l file DEMOgram.l and your .y file DEMOgram.y. Submit these files to Canvas via the Assignment Lab 1.

5 Grading Criterion

Your parser will be run on a number of test DEMO programs, both with and without errors. Your initial score will be based on the percentage of tests that are correctly handled by your parser. Handling all tests correctly would mean an initial score of 100. From there up to 5 points will be added or subtracted based on the quality of your parser's error handling. (If your parser does not detect an exising error, you will not be penalized twice.)

6 Advice

It is next to impossible to debug your parser by entering grammar rules for the whole language and then starting to test it. Instead, you should adopt an incremental approach. Enter the rules for a few grammar productions at a time and test them. When you are convinced that they work, add some more productions and repeat the testing process.

From a practical perspective, it makes sense to begin with the productions for the expression grammar. Once your parser is handling expressions correctly, add assignment, then statement lists. You can then expand to the control flow and I/O statements, declarations, and, eventually, the entire langauge.

Get your parser working properly for correct input programs first. From there, develop error-handling capability for handling multiple errors in a single parse. Extend this capability as much as you have time for.