A Scanner and Parser for DEMO

Programming Assignment 1
COMP 506, Spring 2019

Project Due Date: February 15, 2019 @ 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 and third programming assignments 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;
3. demonstrate the correctness of your scanner and parser by running them on the provided collection of input programs; and
4. write at least two test programs in DEMO that your scanner and parser can accept.

In this project, the output of your scanner 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 linenumber where the error was found.

Your work product for this project is a working scanner / parser for DEMO, written in the C programming language. You will submit a **tar**[1] file that contains the source code, a makefile, and a README file. The README file should explain how to build and execute your scanner / parser.

In the later projects, you will extend your scanner / parser to generate ILOC code that implements and/or optimizes the input DEMO program. You should take care in this project to create a code base on which you can build the later projects.

[1] It must be a **tar** file, not some other kind of archive.

3 Suggested 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` (see, for example, the pages cited on the "Projects" part of the course web site).

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 driver that parses command line arguments and invokes your parser. For this lab, your parser should be named "`demo`". The command line syntax should be

   ```
   demo [-h] [filename]
   ```

   where items in square brackets ([and]) are optional. The optional `-h` flag should print the command line syntax to `stdout` and quit. The optional `filename` argument specifies a file to be parsed.

   If `filename` is missing, your parser should read from `stdin`.

   You may add additional, optional command-line flags.[2] They should be documented in the output produced by the `-h` command-line flag. Your program must still function correctly with a simple command line, such as "`demo test.demo`".

4. 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 token and run the test program over the DEMO test programs. (Your `flex` scanner will eventually need the header file of token types produced by your `bison` parser.)

5. 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. Modify your driver to invoke your `bison` parser. In C, the `bison` parser is invoked by calling `yyparse()`.

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.

   Making effective use of the error token (and the `bison` macros `yyerrok` and `yyclearin`)

[2] For example, you might add a `-d` flag to produce debugging output, or a `-s` flag to produce a listing of variables and symbols found in the input program.

is not easy or obvious. It will take some experimentation and lots of testing with example programs that contain syntax errors. In some cases, you may want to add productions that match a specific error so that you can generate a more precise error message.

The test programs named `error*.demo` are a start. The reference parser has some error detection and reporting built into it.

4 Submitting Your Work

Directions for submission will be given in class and posted on the course web site.

5 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 an 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.