CSc 453 : Programming Assignment 2

CSc 453 : Programming Assignment 2 (Lexical and Syntax Analysis)

Start Date: Thu Sept 17, 2015

Due Date: 11:59 PM, Mon Oct 5, 2015

1. General
   This assignment involves implementing a scanner and parser for C–, using a scanner generator such as lex or flex and a parser generator as such as yacc or bison for this purpose.
   For this assignment, your code should deal with only the lexical and syntax rules of C–. In other words, anything that requires semantic information-i.e., information involving declarations-should be ignored.

2. The Scanner

2.1. General

The scanner should be implemented as a function that returns, each time it is called, either a positive integer indicating what kind of token was found on the input stream, or the value 0 ("end of file") indicating that no further input is available. Note that keywords cannot be used as identifiers.

The values of different kinds of tokens should be defined as macros to simplify the interface between the scanner and parser. For this purpose, it is simplest to define single-character tokens such as (and ; to have the value of the corresponding character constant, e.g., the value of a "left-parenthesis" token will be that of the character constant '('. (The simplest way to do this is to use yacc -d to generate a file y.tab.h that contains the macro definitions, then #include this file into the scanner. Your make file will have to be set up carefully to make this work right.)

2.2. Comments and Whitespace

Comments and whitespace are to be skipped silently. It is an error to encounter an end-of-file inside a comment.

2.3. Errors

The simplest way to deal with lexical errors is to let the parser worry about them. This can be done by simply returning the value of any unrecognized character to the parser.

3. The Parser
   3.1. General
   You should transform your grammar, as necessary, to eliminate conflicts. The "dangling else" shift/reduce conflict will be tolerated, as will shift/reduce conflicts between error productions, but you will be penalized

for any other conflicts. If you encounter conflicts, you may consult the file y.output generated by yacc (invoked with the -v option) for more information.

3.2. Errors

You are to implement error handling and recovery for syntax errors. This does not include errors involving semantic checking (i.e., anything that demands information from declarations), which will be dealt with in the next assignment.

Your program will be expected to deal with errors in a "reasonable" way. Error messages should be specific and should contain enough information (with at least a line number) to allow the user to locate syntax problems. Error recovery should allow your parser to recover gracefully and continue processing the input even after syntax errors are encountered.

4. Invoking Your Program
   Your program will be called compile. It will read from stdin and send all output to stdout. Error messages will be sent to stderr. E.g.:
   cat foo.c | compile
   At this point, your program will act simply as a syntax checker: syntactically correct input will be accepted silently, while syntax errors will give rise to error messages that will be reported to stderr.

Sometimes students find it difficult to resist the the urge to "improve" on this spec by having their compilers read from a file rather than from stdin. Please resist this urge (or, if you can't, be prepared to sacrifice 10 points off your score). I know you can write code to do file I/O; show me that you can read, understand, and follow a spec. :-)

5. Turnin
   You should turn in the sources to your code on lectura. These should include:

The sources and headers for your scanner and parser. If you use tools such as lex, yacc, etc., you should turn in the specifications given to these tools.

A main routine that calls your parser;

A make file called Makefile: this should support at least the command "make compile", whose effect is to create an executable file in the current directory called compile that implements your parser together with auxiliary routines such as the lexical analyser, error handling routines, etc. This command should result in the creation of your scanner and parser from scratch, by invoking the appropriate tools on the input specifications.

Any additional material you wish to turn in. Any documentation or comments may be turned in in a file README.

To turn in your files, use the command

turnin cs453f15-assg2 files

For more information on the turnin command, try man turnin.

Note: The turnin command copies the files submitted into another directory. Because of this, programs that compile and execute without problems in your directory may not work correctly once they are turned in, because of problems with relative path names in include files and make files. Such problems are considered to be sloppiness inappropriate in a graduate course, and are liable to be penalized heavily.

The output of your program will be compared with our output using diff utility (see diff(1)). With the exception of error messages (where the requirements are given above), your output must follow the specification exactly. For this reason it is recommended that you follow the specification, and instructions for turnin, closely.