Project 1: Scanner for the C--Programming Language

V.N. Venkatakrishnan
CS 473
Version 1.0

Introduction

This assignment helps you develop an understanding of lexical analysis. It involves building a scanner for the C--programming language.

Logistics

You must do this homework by yourself. There are two parts to the "hand-in" of this homework. The first will be a scanner for the C--language (details below). The second will be an essay on your experience in doing this homework. Your essay should be at least 10 lines describing your experience with this homework pointing out what you learnt in the homework, your positive outcomes and any difficulties encountered. Also rate the homework difficulty on a scale of 10, with 1 being a very easy homework to 10 being a very tough homework. In your essay, you must also mention the names of any students in this class with whom you had discussions on this homework. (Limit your discussions to problems, not exchanging solutions or code). You can tar your essay along with your homework submission.

For the programming assignment, submission instructions are described in a later section. Any clarifications and revisions to the assignment will be posted on the course blog page on Piazza.

All late homeworks will be subject to the late homework policy that was described in the first lecture.

Handout Instructions

The material needed for this homework is available from the homework page. Everything is in one tar archive called `scanhw.tar`.

Start early!!

Start by copying `scanhw.tar` to a (protected) directory in which you plan to do your work. (Note: It is your responsibility to protect your work; Since you are doing a practical course in developing a compiler, you are expected to keep your work confidential. If you do not know how to set up permissions under Unix, please do so before you begin this homework). Then give the command "`tar xvf scanhw.tar`". This will cause a number of files to be unpacked in the directory:

cmlexer.l: The file you will start with and build a scanner for the C--language. Take a look at this file, and the sample definitions in it.

globals.h: File containing some global constants.

cmparser.tab.h: The file containing definitions for tokens.

Makefile: A makefile that you can use to compile your submission. A makefile is simply a file that contains the rules to compile your project.

All of these programs are compiled to run on Linux machines.

C--Lexical structure description

The rest of this document describes the syntax of the programming language C--. C--is a subset of C containing the essential procedural features but without many of the more complex features such as pointers.

The project for CS 473 is to write a compiler which compiles C--programs. This document contains all the information you need to write a lexical analyzer for C--. The information necessary to complete the project will follow in several other documents.

Please report any bugs, ambiguities, or other problems with this document to `venkat at uic dot edu`.

1 Lexical Components

The first step of compiling a C--program is to convert the input stream of characters into an input stream of tokens, where each token corresponds to a C--lexeme. The lexemes of a C--program are keywords, primitive type names, literals, punctuation, comments, and identifiers. Each of these is discussed in the following sections.

1.1 Notation

The following notation is used in describing the lexemes:

ϵ	represents the empty string
X^*	represents zero or more occurrences of X
X^+	represents one or more occurrences of X
$X^?$	represents one or zero occurrences of X

1.2 Whitespace

Whitespace makes programs easier to read by humans and is not tokenized. The whitespace characters are space, tab, and newline. Your lexical analyzer must ignore whitespace.

1.3 Keywords and Forbidden Words

C--has the following keywords:

```
else  if  int  return  void  while
```

The case of keywords is significant. `if` is a keyword. `If` is not. Additionally, any C keyword which is not a C--keyword is a forbidden word. Forbidden words cannot be used in a C--program. The purpose of this rule is to make it easy to convert C--programs into legal C programs. In many cases, a C--program can be compiled with a C compiler such as `gcc`. The C--forbidden words are:

```
auto      break    case   char      const
continue  default  do     double    for
goto      long     short  register  switch
```

Your lexical analyzer should generate a token `TOK_ERROR` (which can be used by an error handling routine) if any forbidden word appears in a C--program. Note that for purposes of grading, it is enough to report the first error.

1.4 Identifiers

A C--identifier is a letter, underscore character (`_`) or dollar-sign followed by zero or more letters, digits, underscore characters, and dollar signs. In addition, an identifier must not be a keyword or forbidden word. Note that when matching lexemes, the longest match should always be chosen. For example, `ifelse` is an identifier, not the keyword `if` followed by the keyword `else`. Identifiers are also case sensitive, so `Aaa` and `AAa` are different identifiers.

1.5 Comments

Comments are ignored by the lexical analyzer. Therefore, there is no need to return any token corresponding to a comment. C--has two styles of comments:

`/* comment */`	All characters from `/*` until the first occurrence of `*/` are ignored (just like C and C++).
`// comment`	All characters from the `//` until the end of line are ignored.

Note that `//` is ignored when it appears inside of `/*` and `*/`. Hence, the following is a well-terminated comment:

```
/* this is a comment
```

```
     there is a // but it's ignored */
```

1.6 Primitive Types

The names of the C--primitive types are recognized by the lexical analyzer in a manner similar to keywords. These names are:

```
int  void
```

Names of C primitive types which aren't supported in C--are treated as forbidden words. The unsupported types are:

```
byte   double  long  short
float
```

1.7 Literals

There is only one kind of literal allowed in C--.

1.7.1 Integer Literals

An integer literal is `0` or a non-zero base-10 digit followed by zero or more base-10 digits. The value of an integer literal is the standard base-10 interpretation. Some sample integer literals are:

Literal	**Value**
`1273`	1273_{10}
`9`	9_{10}
`10000`	10000_{10}
`0`	0

1.8 Punctuation

The following characters are C--punctuation:

```
(  )  {  }  [  ]  ;  ,
```

1.9 Operators

The following character sequences are C--operators:

```
=   >   <
==  >=  <=  !=
+   -   *   /
```

The following character sequences are C operators which aren't supported in C--:

```
~    ?    :    ++    --    !
&    |    ^    <<    >>    >>>
+=   -=   *=   /=    &=    |=
^=   %=   <<=  >>=   >>>=
```

These operators are forbidden and should trigger a compile error.

1.10 Other Characters

Any input not conforming to the rules in this section is illegal and should generate an error.

Generating your scanner

Once your `cmlexer.l` is ready with some Lex specifications, place it in the same directory as the Makefile, and type `make` on the command prompt. The Makefile will issue the necessary commands to first generate a scanner, and compile it with `gcc` to generate the `cmlexer` executable.

To test the scanner, use a test file that contains several tokens. (There are many such test cases in the `test` subdirectory). Type `./cmlexer ./test.c` to run the lexer on the test file and observe the output.

2 Deliverables

In this homework, you are required to implement a scanner for C--language whose lexical syntax was described above. In addition, you are supposed to complete the `printToken` function implementation, such that it prints the line number, token name and the matched lexeme as suggested in the input file. Your complete scanner must use the token definition from cmparser.tab.h file so that everyone's output looks uniform. You can choose to maintain the line number yourself, which requires you to keep a counter (in the action part of a Lex rule) that needs to be incremented when your scanner encounters new lines , or have Lex do this for you. To have Lex do this for you, look at the example done in class, where the command `%option yylineno` is used.

Your submission must only contain the Lex specification file. To submit, use the URL on the submission page.

Software and other source material:

1. On `oscar.cs.uic.edu` flex is installed.
2. Some test cases will be made available under the `scan` directory in the `test` subdirectory.