Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 1

Lexical Definition

Due Date: 23:59, October 12, 2018

Your assignment is to write a scanner for the C− (C minus) language in **lex**. This document gives the lexical definition of the language, while the syntactic definition and code generation will follow in subsequent assignments.

Your programming assignments are based around this division and later assignments will use the parts of the system you have built in the earlier assignments, That is, in the first assignment you will implement the scanner using **lex**, in the second assignment you will implement the syntactic definition in **yacc**, in the third assignment you will implement the semantic definition, and in the last assignment you will generate Java Bytecode by augmenting your yacc parser.

This definition is subject to modification as the semester progresses. You should take care in implementation that the programs you write are well-structured and easily changed.

1 Character Set

C− programs are formed from ASCII characters. Control characters are not used in the language's definition except '\n' (line feed) and '\t' (horizontal tab).

2 Lexical Definition

Tokens are divided into two classes: tokens that will be passed to the parser and tokens that will be discarded by the scanner (i.e. recognized but not passed to the parser).

2.1 Tokens That Will Be Passed to the Parser

The following tokens will be recognized by the scanner and will be eventually passed to the parser:

Delimiters

Each of these delimiters should be passed back to the parser as a token.

comma	,
semicolon	;
parentheses	( )
square brackets	[ ]
braces	{ }

Arithmetic, Relational, and Logical Operators

Each of these operators should be passed back to the parser as a token.

addition	`+`
subtraction	`-`
multiplication	`*`
division	`/ %`
assignment	`=`
relational	`< <= != >= > ==`
logical	`&& || !`

Keywords

The following keywords are reversed words of C− (Note that the case is significant):

> **while do if else true false for int print const read boolean**
> **bool void float double string continue break return**

Each of these keywords should be passed back to the parser as a token.

Identifiers

An identifier is a string of letters and digits beginning with a letter. Case of letters is **relevant**, i.e. **ident**, **Ident**, and **IDENT** are different identifiers. Note that keywords are not identifiers. Note: Assume that the length of an identifier will not exceed 256.

Integer Constants

- A sequence of one or more digits starts with a **non-zero** digit, e.g., 123, 45, 6.
- A zero digit only, e.g., 0.

Floating-Point Constants

A sequence of one or more digits with a dot (**.**) symbol separating the integral part from the fractional part. Note that the integral part can start with 0s and fractional part can end with 0s. For example, 0.0 is an acceptable case in our language.

Scientific Notations

A way of writing numbers that accommodates values too large or small to be conveniently written in standard decimal notation. All numbers are written like aEb or aeb ('a times ten to the power of b"), where the exponent b is an integer that can start with 0s, and the coefficient a is any real number and also can start with 0s, called the significand. For example: 1.23E4, 1.23E+4, 01.23E-4, 123E04, etc.

String Constants

A string constant is a sequence of zero or more ASCII characters appearing between double-quote (") delimiters. String constants should not contain embedded newlines. A double-quote or back-slash appearing with a string must use '\' to escape. For example, **"aa \"bb \\cc"** denotes the string constant **aa"bb\cc**.

2.2 Tokens That Will Be Discarded

The following tokens will be recognized by the scanner, but should be discarded rather than passing back to the parser.

Whitespace

A sequence of blanks (spaces) tabs, and newlines.

Comments

Comments can be denoted in two ways:

- *C-style* is text surrounded by "`/*`" and "`*/`" delimiters, which may span more than one line;
- *C++-style* is text following a "`//`" delimiter running up to the end of the line.

Whichever comment style is encountered first remains in effect until the appropriate comment close is encountered. For example

```
// this is a comment // line */ /* with some /* delimiters */ before the end
```

and

```
/* this is a comment // line with some /* and
// delimiters */
```

are both valid comments.

Pragma Directives

We typically use a #pragma directive to control the actions of the compiler (or the scanner in this project) without affecting the program as a whole. A pragma directive starts with "#pragma" followed three options: `source`, `token` and `statistic`. `source` turns source program listing on or off, `token` turns token listing on or off and `statistic` turns identifier frequencies listing on or off. By default, all of the options are on. For example, the following statements are pragma directives:

```
#pragma source on
//the pragma directive above turns on source code listing
#pragma statistic off
//the pragma directive above turns off the identifiers statistics
```

Please note that there shouldn't be any characters before or after the pragma directives. But you can have comments after pragma directives.

```
#pragma token on //comments
#pragma source off /* comments */
```

3 Implementation Hints

You should write your scanner actions using the macros `token` and `tokenString`. The macro `tokenString` is used for tokens that return a string as well as a token, while `token` is used for all other tokens. The first argument of both macros is a string. This string names the token that will be passed to the parser. The macro `tokenString` takes a second argument that must be a string. Following are some examples:

Token	Lexeme	Macro Call
left parenthesis	`(`	tokenString("delim", "(");
break	`break`	tokenString("KW", "break");
identifier	`ab123`	tokenString("id", "ab123");
		tokenString("id", yytext);
integer constant	`23`	tokenString("integer", "23");
boolean constant	`true`	tokenString("KW", "true");
less-than	`<`	token("<");

4 What Should Your Scanner Do?

Your goal is to have your scanner print tokens and lines, based on `Source`, `Token` options. If listing option is on, each line should be listed, along with a line number. If token option is on, each token should be printed on a seperate line, surrounded by angle brackets. Your scanner should also reveal the statistical information of the identifiers base on `statistic` option. For example, given following two inputs:

4.1 Success

For input:

```
// print hello world
void main()
{
  int a;
  double b;
  print "hello world";
  a = 1+1;
  b = 1.23;
  if (a > 1){
    b = b*1.23e-1;
  }
}
```

Your scanner should output:

```
1:// print hello world

<KW:void>
<id:main>
<delim:(>
<delim:)>
2:void main()

<delim:{>
3:{

<KW:int>
<id:a>
<delim:;>
4:  int a;

<KW:double>
<id:b>
<delim:;>
5:  double b;

<KW:print>
<string:hello world>
<delim:;>
6:  print "hello world";

<id:a>
<"=">
<integer:1>
<"+">
<integer:1>
<delim:;>
7:  a = 1+1;

<id:b>
<"=">
<float:1.23>
<delim:;>
8:  b = 1.23;

<KW:if>
<delim:(>
<id:a>
<">">
<integer:1>
<delim:)>
<delim:{>
9:  if (a > 1){

<id:b>
<"=">
<id:b>
<"*">
<scientific:1.23e-1>
<delim:;>
10:    b = b*1.23e-1;

<delim:}>
11:  }

<delim:}>
frequencies of identifiers:
main    1
a       3
b       4
```

4.2 Failure

For input:

```
$a = 1; //php-like
```

Your scanner should output:

```
Error at line 1: $
```

When an error occurs, your scanner should print the line number and the unmatched token and exit with code 1. All error messages should be printed to stderr as the following code:

```
fprintf(stderr, "Error at line %d: %s\n", linenum, yytext);
exit(1);
```

4.3 Online scanner

If you have any problems about what the scanner should output, check out the following online scanner, which reads your input and provides you a standard output.

```
http://sslab.cs.nctu.edu.tw/~jrchang/compiler-f16_project1/web/
User name:Compiler-f16
Password:2016compiler
```

5 *lex* Template

This template (`lextemplate.l`) can be downloaded on **e-Campus (E3)** system.

```
%{
#define LIST                { strncat(buf, yytext, sizeof(buf) - strlen(buf) - 1); }
#define token(t)            { LIST; if (Opt_Token) printf("<%s>\n", #t); }
#define tokenString(t, s)   { LIST; if (Opt_Token) printf("<%s:%s>\n", t, s); }
#define MAX_LINE_LENGTH 257
#define MAX_ID_LENGTH 257
int Opt_Source = 1;
int Opt_Token = 1;
int Opt_Statistic = 1;
int linenum = 1;
char buf[MAX_LINE_LENGTH];
%}

%%
"(" {
    tokenString("delim", "(");
}

\n {
    LIST;
    if (Opt_Source)
        printf("%d:%s\n", linenum, buf);
    ++linenum;
    buf[0] = '\0';
}

. { // Unrecognized character
    fprintf(stderr, "Error at line %d: %s\n", linenum, yytext);
    exit(1);
}
```

```
%%

int main( int argc, char **argv )
{
    if (argc >= 2)
    {
        yyin = fopen( argv[1], "r" );
        if ( NULL == yyin ) {
            fprintf( stderr, "Cannot open: %s\n", argv[1] );
            exit(-1);
        }
    }
    else
    {
        yyin = stdin;
    }

    yylex();

    if (Opt_Statistic)
    {
        // Print frequency table
    }

    exit(0);
}
```

6 How to Build and Execute?

```
% lex lextemplate.l
% gcc -o scanner lex.yy.c -lfl

% ./scanner [input file]
```

7 What to Submit?

You should submit the following items:

- Report: a file describing the abilities of your scanner, the platform to run your scanner, and how to run your scanner
- Your lex scanner (.l file)
- a Makefile in which the name of the output executable file must be named '**scanner**' (**Please make sure it works well. TAs will rebuild your parser with this makefile. No further grading will be made if the *make* process fails or the executable '*scanner*' is not found.**)

We suggest you to test your program (including scanner and Makefile, etc.) on the linux workstaion of CS.

8 How to Submit the Assignment?

Create a directory, named ”YourID” and store all files of the assignment under the directory. Zip the directory as a single archive, name the archive as ”YourID.zip”. Upload the zipped file to the **e-Campus (E3)** system.

Note that the penalty for late homework is **15% per day** (weekends count as 1 day). Late homework will not be accepted after sample codes have been posted. In addition, homework assignments must be individual work. If I detect what I consider to be intentional plagiarism in any assignment, the assignment will receive **zero credit**.