Simple Java — Scanner

This assignment requires you to write a Simple Java Scanner by using Lex. Your Scanner must be able to handle all reserved words, symbols, and comments.

If an invalid token is encountered, your scanner must generate an error message (error detection). Your scanner should process as much input data as possible. If an error occurs, your scanner should generate an error message and then continue to process other input data(error recovery).

The Scanner's output will list each token along with its type (integer, float, ID, reserved word, string, operator, symbol, comment), its line number, and the position of its first character. You need to create your own symbol table, which will record all of found Identifiers (IDs) (see rule 6 for details).

1. What to Submit

You must submit the following files:

- ☐ **Scanner, named – your student ID.l**
- ☐ **Your test files**
- ☐ **Makefile**
- ☐ **A Readme.pdf containing:**
  - ☐ **Lex version**
  - ☐ **Operating platform**
  - ☐ **Execution method**
  - ☐ **How you handle the issues in this specification**
  - ☐ **Problems which you have encountered while writing this assignment**
  - ☐ **The results of executing all of test files, saved as an image file.**

Please compress all of the above requirements into one file, named – your student ID_hw1

The following description is a lexical definition of Simple Java.

2 Character Set

Simple Java is composed of ASCII characters. The Simple Java language definition does not use control characters.

3. Lexical Definitions

Tokens are divided into two categories: (1) tokens that will be passed to the parser; (2) tokens that will be directly discarded by the scanner. (Note that they will be recognized but not passed to the parser).

3.1 Tokens Passed to the Parser

Symbols

Each symbol will be returned to the parser as a token.

Comma	,
Colon	:
Semicolon	;
Parentheses	( )
square brackets	[ ]
Brackets	{ }

Arithmetic, Relational, and Logical Operators:

Each operator is sent back to the parser as a token.

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

The following keywords are reserved words for Simple Java (case-sensitive).

boolean, break, byte, case, char, catch, class, const, continue, default, do, double, else, extends, false, final, finally, float, for, if, implements, int, long, main, new, print, private, protected, public, return, short, static, string, switch, this, true, try, void, while ...

Each keyword listed above will be sent back to the parser as a token.

Identifiers

An identifier is a string of characters and numbers, and must begin with a single character. Characters are case-sensitive; for example, \`peter\`, \`Peter\`, and \`PETER\` are three different identifiers. Keywords cannot be used as identifiers. You can use a function name as an identifier.

Integer Constants

Integers are combinations of one or more numbers, and can be positive or negative. For example: 100, -200.

Float Constants

Float Constants can be positive or negative, and can be represented by using either a decimal point or a scientific notation. For example:

- 1.0, 3.14
- 12.25e+6, -2E-2

3.2 Tokens that will be discarded by the scanner

The following tokens will be recognized by the scanner, but they will be discarded directly, instead of being passed to the parser:

Whitespace

space, tabs, newline

Comments

Comments can be represented in two ways:

- C-style: Text enclosed in /* and */ can span multiple lines.
- C++-style: Text consisting of // followed by text cannot span multiple lines.

For example:

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

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

// delimiters */
```

All comments are valid.

4. Recovery

Your scanner must process as much input data as possible. In case of errors, it must be able to recover and continue processing other input data.

5. Symbol Tables

You must implement symbol tables to store all identifiers. Symbol tables should be designed for easy creation and retrieval of data, so hash tables are typically used for implementation. To create and manage symbol tables, at least the following functions must be provided:

- create (): Create a symbol table.
- lookup (s): Return the index of the string S; if string S is not found, returns -1.
- insert (s): Add ‘s’ to the symbol table and return the index of its storage location.
- dump (): Print out all the data in the symbol table.

6. What should your scanner be able to do?

The scanner's output will list each token along with its type (integer, float, ID, reserved word, string, operator, symbol, comment), the line number of the token, and the position of its first character. Finally, you must print all the identifiers in the symbol table. For example, suppose you have the following code:

```
// print hello world
{
print("hello world");
int a = 5 + 5.5;
}
```

Your scanner will output the following result:

```
Line: 1, 1st char: 1, “// print hello world” is a “comment”.
Line: 2, 1st char: 1, “{” is a “symbol”.
Line: 3, 1st char: 3, “print” is a “reserved word”.
Line: 3, 1st char: 8, “(” is a “symbol”.
Line: 3, 1st char: 10, “hello world” is a “string”.
Line: 3, 1st char: 21, “)” is a “symbol”.
Line: 3, 1st char: 22, “;” is a “symbol”.
Line: 4, 1st char: 3, “int” is a “reserved word”.
Line: 4, 1st char: 7, “a” is an “ID”.
Line: 4, 1st char: 9, “=” is a “operator”.
Line: 4, 1st char: 11, “5” is an “integer”.
Line: 4, 1st char: 13, “+” is an “operator”.
Line: 4, 1st char: 15, “5.5” is a “float”.
Line: 4, 1st char: 16, “;” is a “symbol”.
Line: 5, 1st char: 1, “}” is a “symbol”.
The symbol table contains:
a
```

lex sample code

```
%{
#define MAX_LINE_LENG 256
#define LIST strcat(buf,yytext)
#define token(t) {LIST; printf("<%s>\n",t);}
#define tokenInteger(t,i) {LIST; printf("<%s:%d>\n",t,i);}
#define tokenString(t,s) {LIST; printf("<%s:%s>\n",t,s);}
int linenum = 0;
char buf[MAX_LINE_LENG];
%}
%%
"(" {token('(');}
[0-9]+ {tokenInteger(yytext, atoi(yytext));}
\n {
        LIST;
        printf("%d: %s", linenum++, buf);
        buf[0] = '\0';
        }
[ \t]* {LIST;}
. {
        LIST;
        printf("%d:%s\n", linenum+1, buf);
        printf("bad character:'%s'\n",yytext);
        exit(-1);
        }
%%
main(){
yylex();
return 0;
}
```