Introduction to Compiler Design by Prof. Yi-Ping You

Project Assignment 1

Lexical Definition

Due Date: 23:59, March 27, 2014

Your assignment is to write a scanner for the $\mathcal{P}$ 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

$\mathcal{P}$ 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	;
colon	:
parentheses	( )
square brackets	[ ]

Arithmetic, Relational, and Logical Operators

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

addition	`+`
subtraction	`-`
multiplication	`*`
division	`/` **mod**
assignment	`:=`
relational	`< <= <> >= > =`
logical	**and or not**

Keywords

The following keywords are reversed words of $\mathcal{P}$ (Note that the case is significant):

array begin boolean def do else end false for integer if of print read real string then to true return var while

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.

Integer Constants

A sequence of one or more digits. As in the C language, an integer that begins with 0 is assumed to be octal; otherwise, it is assumed to be decimal.

Floating-Point Constants

A sequence of one or more digits with a dot (.) symbol separating the integral part from the fractional part.

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 and the coefficient a is any real number, called the significand. For example: 1.23E4, 1.23E+4, 1.23E-4, 123E4, 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 appearing with a string must be written twice. For example, **"aa""bb"** denotes the string constant **aa"bb**.

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.

Pseudocomments

A special form of *C++-style* comments, called *pseudocomments*, are used to signal options to the scanner. Each pseudocomment consists of a *C++-style* comment delimiter, a character `&`, an upper-case letter, and either a '`+`' or '`-`' ('`+`' turns the option "on" and '`-`' turns the option "off"). In other words, each pseudocomment either has the form `//&C+` or the form `//&C-` where `C` denotes the option.

There may be up to 26 different options (A–Z). Specific options will be defined in the project description. A comment that does not match the option pattern exactly has no effect on the option settings. Undefined options have no special meaning, i.e. such pseudocomments are treated as regular comments. For this project, define two options, `S` and `T`. `S` turns source program listing on or off, and `T` turns token listing on or off. By default, both options are on. For example, the following comments are pseudocomments:

```
//&S+
//&S-
//&S+&S- This leaves the S option on because the rest of the comment is ignored
```

3 Implementation Hints

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

Token	Lexeme	Macro Call
left parenthesis	`(`	token('(');
begin	`begin`	token(KWbegin);
identifier	`ab123`	tokenString(identifier, "ab123");
		tokenString(identifier, yytext);
integer constant	`23`	tokenInteger(integer, 23);
boolean constant	`true`	token(KWtrue);

4 What Should Your Scanner Do?

Your goal is to have your scanner print tokens and lines, based on `S`, `T` 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. For example, given the input:

```
// print hello world
begin
  var a : integer;
  var b : real;
  print "hello world";
  a := 1+1;
  b := 1.23;
  if a > 1 then
    b := b*1.23e-1;
  end if
end
```

Your scanner should output:

```
1:  // print hello world
<KWbegin>
2:  begin
<KWvar>
<id: a>
<:>
<KWinteger>
<;>
3:    var a : integer;
<KWvar>
<id: b>
<:>
<KWreal>
<;>
4:    var b : real;
<KWprint>
<string: hello world>
<;>
5:    print "hello world";
<id: a>
<:=>
<integer: 1>
<+>
<integer: 1>
<;>
6:    a := 1+1;
<id: b>
<:=>
<float: 1.23>
<;>
```

```
7:    b := 1.23;
<KWif>
<id: a>
<>>
<integer: 1>
<KWthen>
8:   if a > 1 then
<id: b>
<:=>
<id: b>
<*>
<scientific: 1.23e-1>
<;>
9:      b := b*1.23e-1;
<KWend>
<KWif>
10:   end if
<KWend>
11:  end
```

5 *lex* Template

This template (`lextemplate.l`) may be found online on the course forum.

lextemplate.l

```
%{
#define LIST                strcat(buf, yytext)
#define token(t)            {LIST; if (Opt_T) printf("<%s>\n", #t);}
#define tokenInteger(t, i) {LIST; if (Opt_T) printf("<%s:%d>\n", #t, i);}
#define tokenString(t, s)  {LIST; if (Opt_T) printf("<%s:%s>\n", #t, s);}
#define MAX_LINE_LENG       256

int Opt_S = 1;
int Opt_T = 1;
int linenum = 1;
char buf[MAX_LINE_LENG];
%}

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

```
int main( int argc, char **argv )
{
  if( argc != 2 ) {
    fprintf( stderr, "Usage: ./scanner [filename]\n" );
    exit(0);
  }

  FILE *fp = fopen( argv[1], "r" );
  if( fp == NULL ) {
    fprintf( stderr, "Open file error\n" );
    exit(-1);
  }

  yyin = fp;

  yylex();

  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
- Makefile (includes all commands to create the scanner)

8 How to Submit the Assignment?

You should create a directory, named "YourID" and store all files of the assignment under the directory. Once you are ready to submit your program, zip the directory as a single archive, name the archive as "YourID.zip", and upload it 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 reduced or, usually, **zero credit**.