README for the scanner project

Wuu Yang
National Chiao-Tung University, Taiwan, Republic of China
July 28, 2022

For the first part of the compiler project, we will implement a scanner. Our target language is minipascal, whose syntax is given in the file 01-minipascal-spec.pdf. Since we will use the lex tool, you will need to prepare the input for lex.

Your work is to study 01-minipascal-spec.pdf and define the tokens used in the minipascal language. If you do not know how to start the project, please study the file SAMPLE1-standard-pascal-no-yacc.l or SAMPLE2-standard-pascal-need-yacc.l.

In particular, you need to add regular expressions for num, strings, and comments.

Comments in a program are marked with two slashes and continue to the end of the current line. For example,

```
a := a + 1 ; // This is a single-line comment.
```

Comments may also be enclosed in the pair /* and */, such as

```
/* hello, this is also a comment.
This is the second line of the comment.
This is the third line of the comment.  */
```

This comments may span multiple lines. Your scanner should print a message for each comments. Comments will not be returned to the parser. There is no nested comments. To handle comments, you will need to use the `BEGIN` feature of lex.

A number could be prefixed with an optional positive/negative sign. A number could be an integer or a floating-point number. They both are considered as the token num. You need to define the regular expression for a number in an appropriate way. You also need to define the scientific notation,

such as 10.3E+5.

Furthermore, in minipascal, there are string constants, such as `"hello"`. A string is enclosed in a pair of double quotes. A string may not contain end-of-lines and double quotes unless they are properly escaped, as in C. On the other hand, there are no char type, no char variables, no char constants. You need to define the regular expressions for string constants in a reasonable way.

You also need to implement two pragmas in the scanner:

```
# pragma list on
# pragma list off
```

A pragma has the # sign on the first column. This pragma turns on/off listing of the source program. The default of listing is on. Pragmas are intended for debugging use. There may be other pragmas in the future, such as `# pragma symtab`.

When the list pragma is turned on, the scanner should print every token and every comment found in the input stream in the following format, where `type` is the token number:

```
0-th token(type:28) on line 1, position 1 : PROGRAM
1-th token(type:39) on line 1, position 9  : mytest
2-th token(type:52) on line 1, position 16  : (
3-th token(type:39) on line 1, position 18  : input
4-th token(type:43) on line 1, position 24  : ,
...
```

The list continues until the list pragma is turned off or until the end of the file.

An identifier `id` begins with an English letter a-z and may include English letters, digits (0-9), and the underscores _. The underscore _ cannot be the last character of an identifier.

Please come to discuss with me if you need help. Do not leave the project blank.

List of Tokens in minipascal

The list of tokens for minipascal includes the following tokens:

```
program
id
(
)
;
.
,
var
:
array
[
]
num
string
stringconst
of
integer
real
function
procedure
begin
end
:= (assignop)
if
```

```
then
else
while
do
<
>
<=
>=
=
!=
+
-
*
/
not
..
and
or
```

Special Issues

1. Note that comments, string constants, and scientific notations are difficult. You may use the BEGIN feature in lex to handle thme.

2. The slash character / means division. The double slashes // indicate the beginning of one-line comments.

3. An input string such as `123.456EBCD` should be scanned as two tokens: `123.456` (which is a num token) and `EBCD` (which is an indentifier). This is not an error in the scanner stage. However, the parser will report this error.

4. A lexical error is usually caused by an illegal character, that is, a character that cannot appear in any token except in a string. For example, `@` is

an illegal character in a C program. When encountering an illegal character, the scanner can simply post an error message, throw away that character, and re-start scanning from the next character.

Tools

The GNU flex distribution is located at ftp://prep.ai.mit.edu/pub/gnu/ You may find the source there. There are also lex/flex for Windows systems. They sever the same purpose.

To generate a scanner, run the following commands:

`flex SAMPLE1-standard-pascal-no-yacc.l`

The above command will generate a C file named "lex.yy.c". Use a C compiler to compile the C file "lex.yy.c".

`cc lex.yy.c -ll`

The above command will produce an executable file, say "a.out". Then execute the scanner:

`a.out scanner-test01.p` (or `./a.out < scanner-test01.p`)

The directory ScannerTestCases contains more test cases.

Handin

For the 1st (scanner) project, you need to turn in your lex files, yacc files (if you have one), semantic routine files (if you have semantic rountines. For the scanner project, you usually do not need semantic routines.), test cases, and the executable code. You also need to prepare a Makefile. The TA will run your program simply with

```
make test
```

You may add a readme.txt file if you want to tell the TA additional details of your scanner.

Please refer to https://github.com/ICD20/hw1 for additional material and a sample Makefile. There is also a docker container that you can use. (The docker container does not work with older versions of Windows, such as Windows 7.)

Put all of the above files in a single zip file which will be named "DDDDDDD-scanner.zip", where DDDDDDD is your student id. Upload the zip to the new e3 platform.

Deadline of the scanner project is October 4, 2022, Tuesday, 23:59 pm.

–