40-414 – Compiler Design, fall 1403, Sharif University of Technology

Programming Assignment I (Scanner)

Released: Tuesday, 01/08/1403

Due: Tuesday, 22/08/1403 at 11:59pm

1. Overview of the Programming Project

Programming assignments I-III will direct you to design and build a **one-pass** compiler for a simplified version of C programming language (i.e., called C-minus). Each assignment will cover one of the main components of the compiler: 1) scanner, 2) parser, 3) intermediate code generator and semantic analyzer. Each assignment will ultimately result in a working compiler phase that can interface with other phases. You will be implementing these projects in **Python**.

For this part of the project, you are required to implement a lexical analyzer (Scanner). You need to describe the set of tokens for C-minus and then draw a DFA for recognizing each token in C-minus programs. You can then combine these DFAs into one unit and use it as a flowchart to write a **Python** program that receives a text file including a C-minus program and outputs the tokenized form of the program (see section 4).

You may use codes from textbooks, with a reference to the used book in your supplementary document. But, using codes from the internet and/or other groups/students in this course are **strictly forbidden** and may result in a fail grade in the course. You may work either individually or in pairs on this assignment and all subsequent programming assignments. (You should already submitted your group in the provided form)

2. Compiler Specification

The compiler that you implement in these assignments must have the following characteristics:

- The compiler includes four main modules for lexical analysis, syntax analysis, semantic analysis, and intermediate code generation.
- The compiler is **one-pass**; the input program is read only once and the four main parts of the compiler work in a pipeline fashion.
- The input to the compiler (through a file called '**input.txt**') is a text file including a C-minus program, which has to be translated.
- In each of the four programming assignments, you implement one of the four main modules of the compiler; ultimately, the outcome of these separate assignments should be combined into one working compiler. The produced code will be tested using a tester interpreter.
- There is a strong dependency between programming assignments I-IV. For instance, programming assignment II is totally dependent on this assignment. In other words, your parser cannot work correctly without a scanner that cannot tokenizes the input program correctly. Similarly, your intermediate code generator will not work without a correct scanner or parser.

3. C-minus

C-minus, a simplified version of the C programming language was introduced in [1]. In this project, we implement a **one-pass** compiler for a slightly modified version of C-minus. In this part of the project, you implement a scanner to recognize the tokens of input C-minus programs.

4. Scanner Results

As it was mentioned above, the scanner receives a text file including a C-minus program and outputs the tokenized form of the program. The scanner must have a function called '**get_next_token**' which after each invocation, proceeds in the input program (character by character) until a valid token is recognized. The function then returns the recognized token as a pair and in the form of the pair:

(Token Type, Token String).

For this assignment, you must also write a while loop which calls 'get_next_token' repeatedly until all tokens of the input C-minus program are recognized. The tokens must be saved in a text file called '**tokens.txt**', where each line has a number that corresponds to a line number in the input file, and a sequence of token pairs that have been recognized on that line. Please note that this while loop is needed only for this assignment. In the next assignment, the 'get_next_token' function will be called by the **parser** whenever a token is required and this while loop will no longer be required.

You can assume that the input includes only ASCII characters (represented in UTF-8). Your scanner should be robust and work for any conceivable input. For example, it should not require that all tokens be separated by a space character. For instance, it should be able to recognize the same set of tokens for both of the following inputs:

```
if(b==3){a=3;}else{b=4;}
if ( b == 3 ) { a = 3 ; } else { b = 4 ; }
```

Your scanner must also handle errors such as an illegal character appearing in the input stream (with the exception of comments, which may include any character), or an ill-formed number such as '125d'. You also should make some preparations for a graceful termination in the case of a fatal error. Uncaught exceptions are not acceptable.
All token types that your scanner must be able to recognize are shown in the table below:

Token Type	**Description**
NUM	Any string matching: [**0-9**]$^{+}$
ID	Any string matching: [**A-Za-z**][**A-Za-z0-9**]*
KEYWORD	**if, else, void, int, while, break, return, endif**
SYMBOL	**; : , [ ] ( ) { } + - * / = < ==**
COMMENT	Any string after // until the end of line (**\n** or **EOF**) Any string between a **/*** and a ***/**[1]
WHITESPACE	blank (ASCII 32), **n** (ASCII 10), **r** (ASCII 13), **t** (ASCII 9), **v** (ASCII 11), **f** (ASCII 12)

[1] Please note that **NUM**, **ID**, and **KEYWORD** tokens can be the very last token of input programs and should be correctly recognized when we reach to **EOF** (End Of File).

4.1. Error Handling

Possible lexical errors are recovered by the **Panic Mode** method. In this method, when a lexical error is detected, the scanner successively deletes input characters of the remaining input until a well-formed token is found. Those characters that have been processed for the current token up to the point that the error has been detected are thrown away, too (see the input-output examples in section 4.4). Lexical errors must be recorded in a text file called '**lexical_errors.txt**'. Each error is reported on a separate line as a pair including a string (i.e., a string of characters that have been thrown away by the scanner) and a message together with the corresponding line number in which the error has occurred. If the input program is lexically correct, the sentence '**There is no lexical error.**' should be written in '**lexical_errors.txt**'.

- When an invalid character (one that can't begin any token) is encountered, a string containing just that character and the message '**Invalid input**' should be saved in 'lexical_errors.txt' Scanner resumes at the following character.
- If a comment remains open when the end of the input file is encountered, record this error with just the message '**Unclosed comment**'. In this type of errors, a long string (i.e., the unclosed comment) might be thrown away by the scanner. However, it is sufficient to print at most the first seven characters of the unclosed comment with three dots. (See the last example in Figure 3).
- If there is a '*/' outside a comment, the scanner should report this error as '**Unmatched comment**', rather than tokenizing it as * and /.
- If you see '125d', you must report this error as '**Invalid number**', rather than tokenizing it as a **NUM** and an **ID**.
- The scanner should recognize tokens with at most one lookahead character. For instance, for recognizing all SYMBOL tokens except '=', there is no need to read any lookahead character. In the case of '=', you need to read just one lookahead character in order to distinguish '=' from '=='.

Note that scanners generally only detect a very limited class of errors (i.e., lexical errors). Thus, do not try to check for errors that are beyond the lexical level. For instance, you must not check whether a variable has been declared before its usage, which is regarded as a semantic error.

4.2. Symbol Table

Programs tend to have many occurrences of the same identifier. Scanners usually store keywords and identifiers in a table called '**Symbol Table**'. For each identifier and/or keyword, there will be a row in the symbol table. It is generally the task of the scanner to add a new row (or record) for each new identifier (i.e., when an identifier is seen for the first time) to the symbol table and store the lexeme of the identifier in that row. Other pieces of information such as the type and arity of the identifiers will be added to the table by other modules (e.g., intermediate code generator). The symbol table is normally initialized by the keywords at the very beginning of the compilation. Your scanner must save its symbol table in an output text file called '**symbol_table.txt**'.

4.3. Other Notes

Your scanner should maintain a variable (e.g., lineno) that indicates which line of the input text is currently being scanned. This feature will aid different compiler modules in printing error messages. This variable is changed whenever the scanner reads a new line symbol **\n** (note that **\f** does not change the lineno. It does not end the one-line comments, either). If the input text is a part of a comment, then it may contain any character. Moreover, there is no need to record or report COMMENT and WHITESPACE tokens.

4.4. Input-Output Samples

Note that your scanner will be tested on a number of test cases that check different aspects of the scanner (e.g., correctness, robustness, error handling, etc.). The following figures show a simple sample of such a test case and the expected output. Note that in lines 9 and 14 of Figure 2 (i.e., 'tokens.txt'), 'e' and '2' have been respectively recognized as a result of the **Panic Mode** error recovery.

lineno	code
1	void main (void) {
2	int a = 0;
3	/* comment1*/
4	a = 2 + 2;
5	a = a - 3;
6	cde = a;
7	if (b /* comment2 */ == 3d) {
8	a = 3;
9	cd!e = 7;
10	}
11	else */
12	{
13	// one line comment
14	b = a < cde;
15	{cde = @2;
16	}} endif
17	return; /* comment 3
18	}

Fig. 1 C-minus input sample (saved in 'input.txt')

lineno	Recognized Tokens
1	(KEYWORD, void) (ID, main) (SYMBOL, () (KEYWORD, void) (SYMBOL,)) (SYMBOL, {)
2	(KEYWORD, int) (ID, a) (SYMBOL, =) (NUM, 0) (SYMBOL, ;)
4	(ID, a) (SYMBOL, =) (NUM, 2) (SYMBOL, +) (NUM, 2) (SYMBOL, ;)
5	(ID, a) (SYMBOL, =) (ID, a) (SYMBOL, -) (NUM, 3) (SYMBOL, ;)
6	(ID, cde) (SYMBOL, =) (ID, a) (SYMBOL, ;)
7	(KEYWORD, if) (SYMBOL, () (ID, b) (SYMBOL, ==) (SYMBOL,)) (SYMBOL, {)
8	(ID, a) (SYMBOL, =) (NUM, 3) (SYMBOL, ;)
9	(ID, e) (SYMBOL, =) (NUM, 7) (SYMBOL, ;)
10	(SYMBOL, })
11	(KEYWORD, else)
12	(SYMBOL, {)
14	(ID, b) (SYMBOL, =) (ID, a) (SYMBOL, <) (ID, cde) (SYMBOL, ;)
15	(SYMBOL, {) (ID, cde) (SYMBOL, =) (NUM, 2) (SYMBOL, ;)
16	(SYMBOL, }) (SYMBOL, }) (KEYWORD, endif)
17	(KEYWORD, return) (SYMBOL, ;)

Fig. 2 Output Sample (saved in 'tokens.txt')

lineno	Error Message
7	(3d, Invalid number)
9	(cd!, Invalid input)
11	(*/, Unmatched comment)
15	(@, Invalid input)
17	(/* comm…, Unclosed comment)

Fig. 3 Error Messages (saved in 'lexical_errors.txt')

no	lexeme
1	break
2	else
3	if
4	endif
5	int
6	while
7	return
8	void
9	main
10	a
11	cde
12	b
13	e

Fig. 4 Sample Symbol Table (saved in 'symbol_table.txt')

5. What to Turn In

Before submitting, please ensure you have done the following:

- It is your responsibility to ensure that the final version you submit does not have any debug print statements and that your lexical specification is complete (every possible input has some regular expression that matches).
- You should submit a file named '**compiler.py**', which at this stage only includes the Python code of your scanner. Please write your **full name(s)** and **student number(s)**, and any reference that you may have used, as a comment at the beginning of your program.
- The responsibility of showing that you have understood the course topics is on you. Obtuse code will have a negative effect on your grade, so take the extra time to make your code readable.
- Your scanner will be tested by running the command line '**python compiler.py**' in Ubuntu operating system using Python interpreter version **3.8**. It is a default installation of the interpreter without any added libraries except for '**anytree**', which may be needed for the next programming assignment. **No other additional Python's library function may be used for this or subsequent programming assignments.** Please do make sure that your scanner is correctly compiled in the mentioned environment and by the given command before submitting your code. It is your responsibility to make sure that your code works properly using mentioned OS and Python interpreter.
- Submitted codes will be tested and graded using several different test cases (i.e., several 'input.txt' files) with and without lexical errors. Your Scanner should read 'input.txt' from the same working directory as your scanner (i.e., 'compiler.py') is placed. Please note that in the

case of getting a compile or run-time error for a test case, a grade of zero will be assigned to your scanner for that test case. Similarly, if the scanner cannot produce the expected outputs (i.e., 'tokens.txt', 'symbol_table.txt', and 'lexical_errors.txt') for a test case, a grade of zero will be assigned to it for that test case. Therefore, it is recommended that you test your scanner on several different random test cases before submitting your code.

- In a couple of days, you will also receive 10 input-output sample files.
- Your scanner will be evaluated by the Quera's Judge System (QJS). These 10 samples will be added to QJS. After the assignment's deadline is passed, three new test cases will be substituted for some the sample cases of the Quera and your scanner will be judged again.
- The decision about whether the scanner function to be included in the 'compiler.py' or a as a separate file such as say 'Scanner.py' is yours. However, all the required files should be reside in the same directory as 'compiler.py'. Therefore, I will place all your submitted files in the same plain directory including a test case and run 'python3 compiler.py'. You should upload your program files ('**compiler.py**' and any other text files that your program may need) to the course page in Quera (https://quera.org/course/18830) **before 11:59 PM, Tuesday, 22/08/1403**.
- Submissions with more than 72 hours delay will not be graded. Submissions with less than that delay will be penalized by the following rule:

  Penalized mark = M * (100 – D) / 100

  Where M = the initial mark of the assignment and D is number of hours passed the deadline.
- It is worth mentioning again that, as it was mentioned in section 2, all the subsequent programming assignments will be dependent on the result of this assignment. **Therefore, it is very important to implement the scanner at this early stage. Otherwise, one has to do this assignment later (without any credit) for the sake of subsequent assignments!**

[1] Kenneth C. Louden, Compiler Construction: Principles and Practice, 1st Ed., PWS Pub. Co., 1997