CST 8152 – Assignment #2

Due Date: prior or on July 17th, 2020

Earnings: 20% of your course grade (Part 1 – 5%, Part 2 – 15%)

Purpose: Building a Lexical Analyzer (Scanner)

Part 1: Regular Expressions, Transition Diagram, Transition Table

In this course you will have a gratifying experience to write the front-end of a compiler for a programming language named ***PLATYPUS***. The ***PLATYPUS*** informal language specification is given in **PlatypusILS_S20** document. In order to write a compiler the informal language specification must be converted to a formal language specification. Since ***PLATYPUS*** is a simple, yet complete, programming language it can be described formally with a context-free grammar (BNF) notation.

The ***PLATYPUS*** grammar have two parts: a lexical grammar and a syntactic grammar. The lexical grammar will define the lexical part of the language: the character set and the input elements such as white space, comments and tokens. In Part 2 of the assignment you will use the lexical grammar to implement a lexical analyzer (scanner). The syntactic grammar for ***PLATYPUS*** has the tokens defined by the lexical grammar as its terminal symbols. It defines a set of productions. The productions, starting from the start symbol <program>, describe how a sequence of tokens can form syntactically correct ***PLATYPUS*** statements and programs. This part of the grammar will be used to implement a syntax analyzer (parser) in one of the following assignments.

You will find the complete lexical and syntactical grammar for the ***PLATYPUS*** language in the document **PlatypusBNFGR_S20**. Read very carefully the informal language specification (**PlatypusILS_S20**) and then see how the informal language specification has been converted to a formal specification using a BNF grammar notation.

Part of the Scanner will be implemented using a Deterministic Finite Automaton (DFA) based on a Transition Table (TT) (see below Part 2). In order to create TT you must complete a few preliminary steps. First, you must convert the lexical grammar into Regular Expressions. Then using the regular expressions you must draw a Transition Diagram (TD). Finally, using the Transition Diagram you can create a Transition Table.

Most of the work is done for you (see **RETDTT_S20** document). In this part you have to write a regular expression for the string literals, complete the TD for the string literal regular expression, and finally modify the TT to accommodate the string literal.

The PLATYPUS language has some design flaws. Try to identify them but do not change the language specification.

Part 2: Implementing a Lexical Analyzer (Scanner)

In Part 1, you have had the pleasure to read and analyze the grammar for the PLATYPUS programming language. In Part 2, you are to create a lexical analyzer (scanner) for the PLATYPUS programming language. The scanner reads a source program from a text file and produces a stream of token representations. Actually, the scanner does not need to recognize and produce all the tokens before next phase of the compilation (the parsing) takes action. That is why, in almost all compilers, the scanner is actually a function that recognizes language tokens and produces token representations one at a time when called by the syntax analyzer (the parser).

In your implementation, the input to the lexical analyzer is a source program written in PLATYPUS language and seen as a stream of characters (symbols) loaded into an input buffer. The output of each call to the Scanner is a single Token, to be requested and used, in a later assignment, by the Parser. You are to use a data structure to represent the Token. Your scanner will be a mixture between transition-table driven (DFA) and token driven scanner. Transition-table driven scanners are easy to implement with scanner generators. In token-driven scanner you have to write code for every token recognition. The token is processed as a separate exceptional case (exception or case driven scanners). They are difficult for modifications and maintenance (but in some cases could be faster and more efficient). You will take the middle road.

Transition-table driven part of your scanners is to recognize only variable identifiers (including keywords), integer literals (decimal constants), floating-point literals, and string literals. To build transition table for those tokens you have to transform their grammar definitions into regular expressions, and create the corresponding transition diagram(s) and transition table. As you already know, Regular Expressions are a convenient means of specifying (describing) a set of strings.

In Part 2 your task is to write a scanner program (set of functions). Three files are provided for you on Brightspace LMS: **token.h**, **table.h**, and **scanner.c** (see **Assignment2PF.zip**). Where required, you have to write a C code that provides the specified functionality. Your scanner program (project) consists of the following components:

platy_st.c – The main function. This program and the test files are provided for you on Brightspace in a separate file (**Assignment2MPTF.zip**). .

buffer.h – Completed in Assignment 1. It contains buffer structure declarations, as well as function prototypes for the buffer structure.
buffer.c – Completed. It contains the function definitions for the functions written in Assignment 1.
token.h – Provided complete. It contains the declarations and definitions describing different tokens. Do not modify the declarations and the definitions. **Do not add anything to that file**.

table.h – Provided incomplete. It contains transition table declarations necessary for the scanner. All of them are incomplete. You must initialize them with proper values. It must also contain the function prototypes for the accepting functions. You are to complete this file. You will find the additional requirements within the file. If you need named constants you can add them to that file.

scanner.c - Provided incomplete. It contains a few declarations and definitions necessary for the scanner. You will find the additional requirements within the file.

The definition of the ***scanner_init()*** is complete and you must not modify it. The function performs the initialization of the scanner input buffer and some other scanner components.

You are to write the function ***malar_next_token()*** which performs the token recognition (***malar*** stands for **m**atch-**a**-**l**exeme-**a**nd-**r**eturn). It "reads" the lexeme from the input stream (in our case from the input buffer) one character at a time, and returns a token structure any time it finds a token pattern (as defined in the lexical grammar) which matches the lexeme found in the stream of input symbols. The token structure contains the token code and the token attribute. The token attribute can be an attribute code, an integer value, a floating-point value (for the floating-point literals), a lexeme (for the variable identifiers and the errors), an offset (for the string literals), an index (for the keywords), or source-end-of file value. The scanner ignores the white space. The scanner ignores the comments as well. It ignores all the symbols of the comment including line terminator. The function consists of two implementation parts: token driven (special case or exception driven) processing and transition table driven processing. You are to write both parts. The tokens which must be processed one by one (special cases or exceptions) are defined in **table.h**. You must build the transition table for recognizing the variable identifiers (including keywords), integer literals, floating-point literals, and string literals.

The scanner is to perform some rudimentary error handling – error detection and error recovery.

<u>Error handling in comments</u>. If the comment construct is not lexically correct (as defined in the grammar), the scanner must return an error token. For example, if the scanner finds the symbol **!** but the symbol is not followed by the symbol **!** it must return an error token. The attribute of the comment error token is a C-type string containing the **!** symbol and the symbol following **!**. Before returning the error token the scanner must ignore all of the symbols of the wrong comment to the end of the line.

Error handling in strings*.** In the case of illegal strings, the scanner must return an error token. The erroneous string must be stored as a C-type string in the attribute part of the error token (not in the string literal table***). If the erroneous string is longer than 20 characters, you must store the first 17 characters only and then append three dots (…) at the end.

***Error handling in case of illegal symbols*.** If the scanner finds an illegal symbol (out of context or not defined in the language alphabet) it returns an error token with the erroneous symbol stored as a C-type string in the attribute part of the token.

Error handling of runtime errors. In a case of run-time error, the function must store a non-negative number into the global variable **scerrnum** and return a run-time error token. The error token attribute must be the string "RUN TIME ERROR: ".

The definition of the ***get_next_state()*** is complete and you must not modify it.

The function ***char_class ()*** must return the column index for the column in the transition table that represents a character or character class (for example, [a-zA-Z], [1-9]). You must complete that function.

Additionally, you have to write the definitions of the accepting functions and some other functions (see *scanner.c*). (You may implement your own functions if needed.)

INPORTANT NOTE:
In the scanner implementation you are not allowed to manipulate directly any of the Buffer structure data members. You must use appropriate functions provided by the buffer implementation. Direct manipulation of data members will be considered a crime ☺ against the functional specifications and will render your Scanner non-working.

INPORTANT NOTE:
You are allowed (but not required) to work on this assignment in teams. If you decide to work in a team, be aware that you will be not allowed to change the team later on, but you can dissolve the team and continue working alone on later assignments. A team can have **two** members **only**. Both members must be officially registered in the course and must have submitted their Assignment #1. **By the end of the first week of the assignment period** each team must send us a notification e-mail with the names (first, last) and the student numbers of the team members. Without a proper and timely notification we will not accept any team work. Each team must submit one assignment only. The .h and .c file headers must contain information about both team members as required by the Assignment Submission Standard.
Additionally, on a **separate page** (team page) containing the name and the student ID# of the team members, each of the team members must give a brief description of the work done by her/him on this assignment (including the names of the functions written by the member). Each and every member must be involved in some coding and testing. Each member must code and test at least three functions (one of them must be an accepting function) and the name of the member must be indicated in the function

header. The function ***malar_next_token()*** may have two authors.
Be aware that all of the conditions above **must be met** in order to have your assignment accepted and marked.

What to Submit (Part 1 and Part 2):

<u>Digital Submission</u>

<u>Compress</u> into a **zip** file the following files: all of the project **.h** files, and **.c** files, all **.pls** files, **the output files produced by your program** using the provided standard test files, and a file that contains your **regular expression for the sting literal and your transition table**. If you have done some additional testing you can include your additional input/output test. If you have worked in a team, you must include a **team page** also in the zip file. Upload the **zip** file on Brightspace LMS (BS). The file must be submitted prior or on the due date as indicated at the top of the assignment. The name of the file must be Your Last Name followed by the last three digits of your student number followed by cA2, and finally, followed by your lab section number (s11, s12 or s13). For example: Rabec007_cA2_s11.zip. If your last name is long, you can truncate it to the first 8 letters. **Teams** must submit one .zip file only. The name of the file must contain both member names and section numbers e.g. Name345_Name123_cA2 *_s11_s12*.zip.

<u>In case of emergency (BS LMS is not working)</u> submit your zip file via e-mail.

The submission must follow the course submission standards. You will find the Assignment Submission Standard as well as the Assignment Marking Guide (**CST8152_ASSAMG)** for the Compilers course on the BS LMS.

Assignments will not be marked if the source files are not submitted on time. Assignments could be late, but the lateness will affect negatively your mark: see the Course Outline and the Marking Guide. All assignments must be successfully completed to receive credit for the course, even if the assignments are late.

Enjoy the assignment. And do not forget that:

"There are two kinds of people, those who do the work and those who take the credit. Try to be in the first group; there is less competition there."

Indira Gandhi

And remember to remember:

Murphy's laws (1 and 2)

1. If anything can go wrong, it will.
2. If there is a possibility of several things going wrong, the one that will cause the most damage will be the first one to go wrong.

O'Toole's commentary on Murphy's laws: "Murphy was an optimist."

Ginsber's Theorems
T1. You can't win.
T2. You can't break even.
T3. You can't even quit the game**.**

CST8152 - Compilers, 24 May 2020, S^R