CST 8152 – Assignment #2

Due Date: prior or on October 25th, 2013

Earnings: 15% of your course grade (Part 1 – 3%, Part 2 – 12%)

Purpose: Building a Lexical Analyzer (Scanner)

Part 1: Writing a Grammar for the PLATYPUS Language

In this course you will have a pleasurable experience to write the front-end of a compiler for a programming language named ***PLATYPUS***. The ***PLATYPUS*** informal language specification is given in **PlatypusILS_F13** document. In order to proceed you need to write a grammar for the language. Since ***PLATYPUS*** is a simple, yet complete, programming language it can be described formally with a context-free grammar (BNF).

The ***PLATYPUS*** grammar will 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 grammar and most of the syntactic grammar for the ***PLATYPUS*** language in the document **PlatypusLGR_F13**. There are some “small” errors in both parts of the grammar. Additionally, some of the productions in the syntactic part of the grammar are incomplete. They are indicated with the phrase ***<complete this production>***.

Your task in this part of the assignment is find and correct the “small errors” and to complete the incomplete productions in the syntactic part of the grammar for the ***PLATYPUS*** language. In order to accomplish those tasks, you must first read very carefully the informal language specification (**PlatypusILS_F13**). The completed grammar must define the same language as the language described by the informal language specification.

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

What to Submit (Part 1):

Hand in, on paper, the corrected and completed BNF grammar for the PLATYPUS language. **Remove all explanations and submit only the productions**. All materials must be in printed form. They must have **your name on the top of each page**. Hand-written or anonymous materials will not be accepted for evaluation.

Part 2: Implementing a Lexical Analyzer (Scanner)

In Part 1, you had the pleasure to read, analyze, and complete the grammar for the PLATYPUS programming language. In Part 2, you are to create a lexical analyzer (scanner) for this language. This assignment implements and tests the foundation of your lexical analyzer: **The Scanner**. 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 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 structure to represent the Token. Your scanner will be a mixture between transition-table driven and token driven scanner. Transition-table driven scanners are easy to implement with scanner generators. In token-driven scanner every 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 and octal constants), and floating-point literals. To build transition table for those three 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 certain simple set of strings.

Task 1:

The first task is to write regular expressions, and using them to create a transition diagram and a transition table for the PLATYPUS language variable identifiers (including keywords), integer literals (decimal and octal), and floating-point literals.

Task 2:

The second task is to write a scanner program (set of functions). Three files are provided for you on the web site: **token.h**, **table.h**, and **scanner.c** (see **Assignment2PF.zip**). The main program and the test files will be posted later. 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 BB.

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.

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.

You are to write the function ***mlwpar_next_token()*** which performs the token recognition (***malpar*** stands for **m**atch-a-**l**exeme-**w**ith-a-**p**attern-**a**nd-**r**ecognize) . It "reads" the lexeme from the input stream (in our case from an 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), or an index (for the keywords). The scanner ignores the white space. The scanner ignores the comments as well. It ignores all the symbols that follow the comment beginning prefix **!<** (the **!** symbol followed by a less-than sign) to the end of the line inclusive. 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 above and in **table.h**. You must build the transition table for recognizing the variable identifiers (including keywords), integer literals, and floating-point 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 (as defined in the grammar) correct, the scanner must return an error token. For example, if the scanner finds only a symbol **!** not followed by **<** it returns an error token. The C-type string attribute of the comment error token is the **!** and the first symbol following **!**.

<u>*Error handling in strings*</u>. 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 runtime error, the function must store a non-negative number into the global variable **scerrnum** and return an 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-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 alone, you cannot switch to a team-work later. 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. **By the end of the first week** of the assignment period each team must send me a notification e-mail with the names (first and last) of the team members. Without a proper and timely notification I will not accept any team work. Each team must submit one assignment only. The envelope label and the cover page must contain information about both 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). The page must be signed by the members of the team. 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 ***mlwpar_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):

For Part 1

Refer to Part 1 of the Assignment..

For Part 2

Hand in, on paper:

1. The **regular expressions**, the **transition diagram**, and the **transition table** for the recognition of the PLATYPUS language variable identifiers, integer literals, and floating-point literals.
   All materials must be in a printed form except for the drawings. They must have your **name** (or the team **names**) on the top of each page. Anonymous materials will not be evaluated.
2. The fully documented source listing of the modified **scanner.c** and **table.h** source files containing the Scanner and its supporting declarations, definitions, and functions.
3. The test results from testing your Scanner with the provided test files and main program. You can add your own test files and test results.
4. **The marking sheet for the assignment with your name filled in**.

Digital Drop Box Submission

Compress into a **zip** file the following files: all **.h** files, all **.c** files, all **.pls** files, and your output test files. Include your additional input/output test files if you have any and they are not too large (MB). Upload the **zip** file on Blackboard. The file must be submitted prior or on the due date as indicated in 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. For example: Brown345_cA2.zip. If your last name is long, you can truncate it to the first 5 letters. **Teams** must submit one .zip file only. The name of the file must contain the names of both members e.g. Brown345_Fox123_cA2.zip.

In case of emergency (BB is not working) submit your zip file via e-mail.

Make sure all printed materials (and eventually the disk) are placed into an unsealed envelope and are deposited into the assignment box prior to the end of the due date. If you are late, you must submit the assignment envelope directly to the professor. 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 BB..

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, September 25, 2013, S^R