CST 8152 Compilers – Assignment #4 – The Parser

Assignment Due Date: prior to or on **December 8, 2016**

NOTE: This Assignment cannot be late more than 1 week (see Bonus) and it MUST be a Bug Free Working Compiler

Earnings: 14% of your total grade (Part 1 – 14, Part 2 – Bonus – up to 10)

Purpose: Implementing a Parser and more…

Assignment #4 consists of two parts. Part 1 is mandatory. It involves the PLATYPUS Parser implementation. Part 2 is an optional bonus for those of you who would like to have some challenge and would want to earn some additional marks (up to 10%). Part 2 includes some rudimentary semantic analysis, writing a PLATYPUS cross-compiler, or writing a PLATYPUS interpreter.

Part 1. Implementing a PLATYPUS Parser (14 marks)

You are cordially invited to (in other words, you have to) write a ***Recursive Descent Predictive Parser*** (**RDPP**) for the **PLATYPUS** language. You are to integrate the Parser with your existing lexical analyzer and symbol table in order to complete the front-end of your PLATYPUS compiler. The implementation is broken into two tasks.

Task 1. Modifying the Grammar (4 marks).

In order to build a RDPP you need to modify the **syntactical part** of the PLATYPUS Grammar (The Platypus Syntactic Specification). The grammar provided for you in ***PlatypusLGR_F16,doc*** is an LR grammar (that is, a grammar suitable for LR parsing). You must transform it into an LL grammar suitable for ***Recursive Descent Predictive Parsing***. To accomplish that you should follow the steps outlined bellow.

1. Check the PLATYPUS Grammar for completeness and correctness.
2. Eliminate the ***left recursion*** and apply ***left factoring*** where needed.

Some of the syntactic productions must be rewritten to make them suitable for recursive decent predictive parsing. Do not forget that our grammar (***PlatypusLGR_F16.doc***) is an LR grammar, which must be transformed into an equivalent LL grammar. For example, the productions of the type

<statements> -> <statement> | <statements> <statement>

must be rewritten in a convenient for transformation form as

<statements> -> <statements> <statement> | <statement>

Once you rewrite the production, you must eliminate the immediate left-recursion. The transformation will produce the following two new productions:

```
 <statements> -> <statement> <statements’>
 <statements’> -> <statement> <statements’> | ε
```

In some cases it is possible to rework the grammar to avoid some of the transformations. This approach is often applied to production, which contain a “left-factor.” For example, the output statement

```
<output statement> ->
          OUTPUT (<opt_variable list>);
        | OUTPUT (<string literal>);
```

can be reworked in the following way:

```
<output statement> ->
          OUTPUT (<output list>);

<output list > -> <opt_variable list> | STR_T
```

where STR_T is a string literal token produced by the scanner.

The <output statement> production does not contain a left-factor anymore.

It is also possible to simplify the grammar applying formal simplification procedures (see pages 223-224 (old 183-185) in your textbook). Do not simplify the grammar for this assignment.

3. Build the FIRST set for the **syntactic** grammar.

After you transform the grammar you must build the **FIRST set** for each of the grammar productions (nonterminals). A FIRST set for a production (nonterminal) is a set of terminals that appear as left-most symbols in any of the production alternatives. When you build the FIRST set, if some of the elements of the set are non-terminals, they must be replaced by their FIRST sets and so on. The final FIRST set must contain only **terminals** (tokens). The elements of the set must be **unique**. This is essential for the predictive parser. If they are not unique, the left factoring transformation must be applied to the corresponding production.

4. **Part 1 - Task 1 Submission:** Type and print the ***entire syntactic grammar*** and the corresponding FIRST sets. Do not remove the original productions. If a production is to be corrected or transformed, write the **modification** below the original production and **indicate clearly** the type of the changes applied to the original production. Do not include the provided explanatory text (only the productions). Print your name on every page.

Now you are ready for the next task. You can work on both tasks simultaneously - production by production and function by function. I strongly recommend this approach.

Task 2. Writing the Parser(10 marks)

To build the RDPP follow the steps outlined below.

Step 1:

Name your RDPP source code file ***parser.c***. Create ***parser.h***. Include the required system and user header files. Define two static global variables: ***lookahead*** of type ***Token***, and ***sc_buf*** of type pointer to ***Buffer***. Additionally define a global variable **synerrno** of type ***int***. You may add additional variable declarations and constants definitions, if necessary (and it is). Then declare the functions used in the parser implementation. All function prototypes, variable definitions/declarations, and constant definitions must be in ***parser.h***.

Step 2:

Write your ***parser()*** function.

```
void parser(Buffer * in_buf){
  sc_buf = in_buf;
  lookahead = mlwpar_next_token(sc_buf);
  program(); match(SEOF_T,NO_ATTR);
  gen_incode("PLATY: Source file parsed");
}
```

Step 3:

Write your ***match()*** function. The prototype for the function is:

```
void match(int pr_token_code,int pr_token_attribute);
```

The ***match()*** function matches two tokens: the current input token (***lookahead***) and the token required by the parser. The token required by the parser is represented by two integers - the token code (***pr_token_code***), and the token attribute (***pr_token_attribute***). The attribute code is used only when the token code is one of the following codes: **KW_T**, **LOG_OP_T**, **ART_OP_T**, **REL_OP_T**. In all other cases the token code is matched only.
If the match is successful and the ***lookahead*** is **SEOF_T**, the function returns.
If the match is successful and the ***lookahead*** is not **SEOF_T**, the function advances to the next input token by executing the statement:

```
lookahead = mlwpar_next_token (sc_buf);
```

If the new lookahead token is **ERR_T**, the function calls the error printing function ***syn_printe()***, advances to the next input token by calling ***mlwpar_next_token ()*** again, increments the error counter ***synerrno***, and returns.
If the match is unsuccessful, the function calls the error handler ***syn_eh(pr_token_code)*** and returns.

Note: Make your ***match()*** function as efficient as possible. This function is called many times during the parsing. The function will be graded with respect to design and efficiency.

Step 4:

Write the error handling function ***syn_eh()***. This function implements a simple panic mode error recovery.

```
void syn_eh(int sync_token_code)
```

First, the function calls ***syn_printe()*** and increments the error counter. Then the function implements a panic mode error recovery: the function advances the input token (***lookahead***) until it finds a token code matching the one required by the parser (***pr_token_code*** passed to the function as ***sync_token_code***).

It is possible, when advancing, that the function can reach the end of the source file without finding the matching token. To prevent from overrunning the input buffer, before every move the function checks if the end of the file is reached. If the function looks for ***sync_token_code*** different from **SEOF_T** and reaches the end of the source file, the function calls ***exit(synerrno)***.
If a matching token is found and the matching token is not **SEOF_T**, the function advances the input token one more time and returns. If a matching token is found and the matching token is **SEOF_T,** the function returns.

Step 5:
Write the error printing function ***syn_printe()*** .

```
void syn_printe()
```

Note: This function implementation is provided for you in Assignment4MPTF.zip.

The function prints the following error message:

```
PLATY: Syntax error: Line: line_number_of_the_syntax_error
*****  Token code:lookahead token code Attribute: token attribute
```

and returns. For example:

```
PLATY: Syntax error:  Line: 2
*****  Token code: 13 Attribute: NA
PLATY: Syntax error:  Line: 8
*****  Token code:  9 Attribute: 0
PLATY: Syntax error:  Line: 9
*****  Token code:  2 Attribute: sum
PLATY: Syntax error:  Line: 11
*****  Token code:  4 Attribute: 0.5
PLATY: Syntax error:  Line: 17
*****  Token code:  6 Attribute: Result:
PLATY: Syntax error:  Line: 21
*****  Token code: 16 Attribute: ELSE
```

If the offending token is a keyword, variable identifier or string literal you **must** use the corresponding token attribute to access and print the lexeme (keyword name, variable name, or string).
For example, to print the keyword lexeme you must use the ***kw_table*** defined in ***table.h***. **Important note:** You are not allowed to copy the keyword table in *parser.h* or *parser.c*. You must use a proper declaration to create a link to the one defined in *table.h*.
Similarly, you must use the symbol table or the string literal table to print the variable names or the sting literals.

Step 6:

Write the **gen_incode*()*** function. In Part 1 of this assignment the function takes a string as an argument and prints it. Later the function can be modified and used to emit intermediate (Bonus 1) or machine code. The function may be called any time a production is recognized (see ***parser()***). The format of the message is: “PLATY: Program parsed”, “PLATY: Assignment statement parsed”, and so on (see the sample output files).

Step 7:

For each of your grammar productions write a function named after the name of the production. For example:

```
void program(void){
match(KW_T,PLATYPUS);match(LBR_T,NO_ATTR);opt_statements();
match(RBR_T,NO_ATTR);
gen_incode("PLATY: Program parsed");
}
```

Writing a production function, follow the substeps below.

Step 7.1:

To implement the Parser, you **must use** the modified grammar (see Task 1). Before writing a function, analyze carefully the production. If the production consists of a single production rule (no alternatives), write the corresponding function without using the FIRST set (see above). You can use the ***lookahead*** to verify in advance whether to proceed with the production or to call ***syn_printe()*** function. If you do so, your output might report quite different syntax errors than my parser will reports.

Example: The production:

```
<input statement> ->
     INPUT (<variable list>);
```

MUST be implemented as follows:

```
void input_statement(void){
   match(KW_T,INPUT);match(LPR_T,NO_ATTR);variable_list();
   match(RPR_T,NO_ATTR); match(EOS_T,NO_ATTR);
   gen_incode("PLATY: Input statement parsed");
}
```

AND MUST NOT be implemented as shown below:

```
void input_statement(void){
  if(lookahead.code == KW_T
     && lookahead. attribute. get_int== INPUT) {
   match(KW_T,INPUT);match(LPR_T,NO_ATTR);variable_list();
   match(RPR_T,NO_ATTR); match(EOS_T,NO_ATTR);
   gen_incode("PLATY: Input statement parsed");
  }else
   syn_printe();
}
```

This implementation will “catch” the syntax error but will prevent the ***match()*** function from calling the error handler at the right place.

Step 7.2:

If a production has more than one alternatives on the right side (even if one of them is empty), you must use the FIRST set for the production.

For example, the FIRST set for the <opt_statements> production is: { **KW_T** (but not **PLATYPUS, ELSE, THEN, REPEAT**), **AVID_T, SVID_T**, and ε}.

Here is an example how the FIRST set is used to write a function for a production:

```
void opt_statements(){
/* FIRST set: {AVID_T,SVID_T,KW_T(but not … see above),e} */
switch(lookahead.code){
   case AVID_T:
   case SVID_T: statements();break;
   case KW_T:
   /* check for PLATYPUS, ELSE, THEN, REPEAT here and in
   statements_p()*/
    if (lookahead. attribute. get_int != PLATYPUS
         && lookahead. attribute. get_int != ELSE
         && lookahead. attribute. get_int != THEN
         && lookahead. attribute. get_int != REPEAT){
     statements();
     break;
    }
   default: /*empty string – optional statements*/ ;
     gen_incode("PLATY: Opt_statements parsed");
 }
}
```

Pay special attention to the implementation of the empty string. If you do not have an empty string in your production, you must call the ***syn_printe()*** function at that point .

IMPORTANT NOTE: You are not allowed to call the error handling function ***syn_eh()*** inside the production functions and you are not allowed to advance the ***lookahead*** within the production functions as well. Only ***match()*** can call ***syn_eh()***, and only ***match()*** and ***syn_eh()*** can advance ***lookahead.***

ANOTHER NOTE: Each function must contain a line in its header indicating the production it implements and the FIRST set for that production (see above).

Step 8:

Build your parser incrementally, function by function. After adding a function, test the parser thoroughly. You may need some empty functions (stubs) to do the testing for more complex productions.

Part 2. Optional Bonus (up to 10 marks)

Bonus Due Date: prior to or on **December 16, 2016** by 12:00 PM (noon)

Bonus 1 - Semantic Analysis in the Parser (4 marks)

The PLATYPUS language specification stipulates that the default data type of an arithmetic variable can be changed by assigning to it a constant of different data type. The type can be changed only once in a program. Since this rule is not built in the grammar you must implement some kind of semantic analysis. It can be done employing the following strategy. When the parser enters the "assignment statement" function, create a stack and push all the incoming tokens on the stack. When an end of statement is matched, call a function that will analyze the contents of the stack. If a "constant assignment" is found, change the type of the variable and its initial value in the symbol table using the symbol table update functions. This implementation might require some modification of the ***match()*** function.

You can develop a different strategy to solve the problem.

Bonus 2 – PLATYPUS Cross-compiler (7 marks)

Write a PLATYPUS cross-compiler. Your compiler (parser) must generate a complete ANSI C program, which when compiled must produce exactly the same results as the originating PLATYPUS program given the same input data. This task is relatively easy. Use the ***gen_incode()*** function to generate C-code to a file. This implementation might require some modification of the ***main()***, ***gen_incode***, and the ***match()*** functions.

Bonus 3 – PLATYPUS Interpreter (10 marks)

Write a PLATYPUS Interpreter. This task is more difficult than the task above. You need to add some new data structures (stacks and lists), but again with some modifications to the parser you can turn it into an interpreter. Those of you who decide to implement this option should talk to me. I will provide them you with some information about how to build the interpreter.

NOTE: In order to earn the optional bonus marks you must:
1) Have a **working** PLATYPUS compiler project as specified by Assignment 4**.**
2) Submit your bonus program ***separately*** from the required assignment submission in a folder named **bonus1, bonus2,** or **bonus3**.
3) Explain briefly in writing which option you implemented and how you implemented it.
4) Document thoroughly your implementation code. You do not need to print the code.
5) Provide your own test file(s) and an explanation how to use them to prove that your implementation "works."

What to Submit for Part 1 (Task 1 and Task 2):

Hand in, on paper (**no later than 16 December 2016, 12:00 o'clock (noon)**):

1. The modified syntactic part of the PLATYPUS grammar and the FIRST set in printed form. **Indicate** clearly all the changes to the grammar. **Indicate** what kind of transformations you have used to modify the grammar. Remember that the rewritten grammar must be ***equivalent*** to the original one. Do not make your PLATYPUS a SPLATYPUS (S for Sad).

2. The fully documented (file headers, function headers, comments) source listings of your ***parser.h*** and ***parser .c*** files. The function headers of the parser functions should contain only the following: **the name of the production the function implements, author (only if you work in a team), and the FIRST set for the production.**

3. Your test results from testing the parser with the provided test files and main driver ***platy.c***

4. In the cover page of the assignment, you must additionally include an e-mail address at which I can reach you during the exam week and in the first 10 days after the end of the semester. Do not forget this information. It could be crucial for your success in the course.

5. The marking sheet for the assignment with your name filled in

Digital Submission

Compress into a **zip (not rar)** 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. Your bonus (if any) must be in a separate folder. Upload the zip file on Blackboard using the Assignment 4 link in the Assignments folder 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 cA4. For example: Brown345_cA4.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_cA4.zip.

Make sure all printed materials 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 notify the professor by e-mail that the assignment envelope has been submitted.

The submission MUST follow the Assignment Submission Standard. The assignment can be either submitted directly to the professor, or deposited into the assignment box prior to or on the due date. This assignment **cannot be late more than a week –** see the Marking Sheet for this assignment.

Important Note: You are still allowed to work on this assignment in teams. If you have already worked in a team, you can work alone, but you are not allowed to ***change*** the team. If you have worked alone, you are not allowed to work on this assignment in a team. If you work on the bonus as a team, the bonus mark will be divided by 2.
In the cover page you must include additionally an e-mail address at which I can reach you during the exam week and in the first 10 days after the end of the semester. Do not forget this information. It could be crucial for your success in the course.

THE LAST IMPORTANT NOTE: All assignments must be completed to receive credit for the course (see Course Outline).

Enjoy the assignment and do not forget that:

"*It is better to know some of the question than all of the answers*." James Thruber

And remember to remember:

"If you think that education is expensive, try ignorance" Derek Bok

and also

(Please see next page)

Write and sing:

Write in C ("Let it Be")

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."

As the deadline fast approaches,
And bugs are all that I can see,
Somewhere, someone whispers:
"Write in C."

Write in C, Write in C,
Write in C, oh, Write in C.
COBOL's old and blurred,
Write in C.

I used to write a lot of FORTRAN,
For science it worked flawlessly.
Try using it for graphics!
Write in C.

When I spend an endless night,
Debugging some assembly,
Something, somewhere tells me:
"Write in C."

Write in C, Write in C,
Write in C, yeah, Write in C.
BASIC's not the answer.
Write in C.

These days Java came to town.
It perfectly compiles to code named B,
But it’s sometimes slow when turns around.
Write in C.

Write in C, Write in C
Write in C, oh, Write in C.
Pascal won't quite cut it.
Write in C.

A fake C sharps around
trying to bring Java down.
It’s sharp but can’t cut a stone,
so please live C alone.

Write in C, Write in C
Write in C, oh, Write in C.
C is sharp and Sharp isn’t it.
Write in C.

☺ *Your turn here, add a verse or two* ☺
(see Write_in_C_Poems.zip)

Professor: Svillen Ranev
CST8152 - Compilers, F16