BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SY

Group No. 2

Compiler Construction (CS F363)
II Semester 2019-20
Compiler Project (Stage-1 Submission)
Coding Details
(February 24, 2020)

---

1. IDs and Names of team members

ID: **2017A7PS0093P** Name: **AYUSH JAIN**

ID: **2017A7PS0025P** Name: **BHARAT BHARGAVA**

ID: **2017A7PS0117P** Name: **SATVIK GOLECHHA**

2. Mention the names of the Submitted files :

1 bool.c
2 bool.h
3 driver.c
4 enum.h
5 grammar.txt
6 hash.c
7 hash.h
8 keywords.txt
9 lexer.c
10 lexer.h
11 Makefile
12 parser.h
13 remove_comments.c
14 parser.c
15 t1.txt
16 t2.txt
17 t3.txt
18 t4.txt
19 t5.txt
20 t6.txt

3. Total number of submitted files: **20** (All files should be in **ONE folder** named exactly as Group_#, # is your group number)

4. Have you mentioned your names and IDs at the top of each file (and commented well)? (Yes/ no): **YES** [Note: Files without names will not be evaluated]

5. Have you compressed the folder as specified in the submission guidelines? (yes/no): **YES**

6. Lexer Details:

[A]. Technique used for pattern matching: Implementing the **DFA** for the regular expressions of the tokens as specified in the language specification, first on paper and then obtaining the results through code.

[B]. DFA implementation (State transition using switch case, graph, transition table, any other (specify): State transition using **switch case**. Small sub-cases and cases related to a range of values handled using if-else.

[C]. Keyword Handling Technique: **HASHING**

[D]. Hash function description, if used for keyword handling: We've implemented a hash function **of our own**, in which collisions are being resolved by chaining, to learn about the minutest of the implementation details which a compiler requires.

[E]. Have you used twin buffer? (yes/ no) : **YES**

[F]. Lexical error handling and reporting (yes/No): **YES**

[G]. Describe the lexical errors handled by you: 1. Length of an identifier cannot **exceed 20**; it will be reported as an error. 2. Entities such as 20.eab, 10., 5.78Ea3 etc. will not be tokenized fully and a lexical error will be reported. 3. All symbols, the transitions for which enter in a **trap state** in the DFA, will be treated as invalid, and lexical error will be thrown (e.g., '.'). 4. Symbols such as apostrophe (ASCII code 39), tab (ASCII code 09), space (ASCII code 32) etc. are ignored and act like delimiters. Further changes might be made like reporting a few of them as lexical errors, depending upon additional requirements.

[H]. Data Structure Description for tokenInfo (in maximum two lines): It is a 3-tuple returned by the lexer to the parser, consisting of the lexeme, the corresponding token and the line number from where that lexeme is fetched from the source code.

[I]. Interface with parser: **Parse table**. The lexer will return the 3-tuple one-by-one as described above. Through appropriate match from the parse table, the parser will decide the sequence of rules required to write the program and hence parse the stream of tokens.

7. Parser Details:

[A]. High Level Data Structure Description (in maximum three lines each, avoid giving C definitions used):

i. Grammar: The structure includes an **array of pointers**, with each entry of the array pointing to a **rule, a linked list of terminals and non-terminals**. For example, if A -> B c D e is a rule, with the uppercase letters represented as non-terminals and the lowercase letters represented as terminals, then some entry of the array will point to A, i.e., the only non-terminal present on the LHS of the rule. A separate field keeps track of the number of rules in the grammar.

ii. Parse table: The parse table is a **2-D array**, with rows indexed by the enumeration of non-terminals and columns indexed by the enumeration of terminals. Each entry in the parse table, if valid, corresponds to the **rule which is to be picked up for parsing** (for non-terminal A and terminal b, if, say, the value of **table[A,b]** is **t**(≥ 0, i.e., valid rule), then we need to pick the rule number "t" (rule number "t" in grammar file) and continue parsing from there on. However, if t=-1, it is to be reported as a parsing error).

iii. Parse tree: (Describe the node structure also): The parse has been constructed taking into consideration the corresponding entries in the parse table for the terminal and the non-terminal under investigation. Each node includes the **data field, the field corresponding to the number of children and fields serving as the pointers to its children**. The data field includes the lexeme 3-tuple, the pointer to the respective

parent's node and the flag denoting whether it is a leaf node (terminal) or not as its chief constituents. This will be synchronized with appropriate pushes and pops on the stack for parsing.

iv. Parsing Stack node structure: The stack has been implemented as a **linked list** of nodes, with each node holding the enumerated value of a terminal/non-terminal. It further includes pointers to other stack nodes.

v. Any other (specify and describe): We computed the first and follow sets and stored them in **boolean arrays**, instead of applying bitwise operators on some 64-bit long integer, on account of scalability. This means, our number of terminals or non-terminals is **independent of the limitations of the underlying architecture**.

[B]. Parse tree

i. Constructed (yes/no): **YES**

ii. Printing as per the given format (yes/no): **YES**

iii. Describe the order you have adopted for printing the parse tree nodes (in maximum two lines): We've used **n-ary in-order traversal** for printing the parse tree nodes. We'll first recursively traverse the leftmost child of the non-terminal node, followed by the non-terminal node itself and then we'll recursively traverse the remaining children of the non-terminal node. E.g., for the rule A -> B C D, the order of traversal would be B -> A -> C -> D, with each non-terminal involving a recursive in-order traversal in itself.

[C]. Grammar and Computation of First and Follow Sets

i. Data structure for original grammar rules: **ARRAY OF POINTERS TO LINKED LISTS.** Each grammar rule is represented as a linked list as follows: the rule A -> B c D will be represented as A -> B-> c -> D -> (null). Let's say, this is the rule number "r" (counting starts from 0, so the first rule in the grammar file implies rule number 0), so the r^{th} index of the array will store the address of the non-terminal on the LHS of the production rule. In the example, the uppercase letters A, B and D are assumed to be non-terminals, while "c" is a terminal (token). It is compulsory to have a single non-terminal on the LHS of the production rule, while RHS can be a string of tokens and non-terminals.

ii. FIRST and FOLLOW sets computation automated (yes /no): **YES**

iii. Data structure for representing sets: **BOOLEAN ARRAYS**

iv. Time complexity of computing FIRST sets: **O(n)** in number of tokens and non-terminals

v. Name the functions (if automated) for computation of First and Follow sets: **populate_first(), populate_follow(), construct_first_follow_set()**

vi. If computed First and Follow sets manually and represented in file/function (name that): N/A

[D]. Error Handling

i. Attempted (yes/ no): **YES**

ii. Printing errors (All errors/ one at a time): Prints all errors at a time. Even distinguishes between lexical and syntax errors.

iii. Describe the types of errors handled: The errors handled are **lexical errors**, obtained during the lexical phase, and **syntax errors** obtained during parsing. Lexical errors arise from an incorrect lexeme, i.e., a lexeme not meeting the requirements of the regular expressions of any of the tokens specified in the language specification. Syntax errors arise from a failed attempt to parse the stream of tokens supplied by the lexer completely, which primarily includes errors due to missing token/errors due to extra token somewhere. Note that symbols like apostrophe (‘), space, tab etc. have been assumed as being ignored by our language.

iv. Synchronizing tokens for error recovery (describe): We have used panic mode error recovery for dealing with syntax errors. More specifically, we’ve dealt them in 2 cases. In the first case, if the top of the stack is a terminal, we ignore the input stream of tokens until that particular terminal is returned by the lexer. In the second case, if the top of the stack is a non-terminal, we keep skipping the tokens until we receive a valid entry in the parse table. The main idea behind this approach keeps in mind the **need for re-continuing the parsing process** as soon as possible.

v. Total number of errors detected in the given testcase t6(with_syntax_errors).txt: **22** errors reported (for a total of 9 syntax errors and 2 lexical errors). The lexical errors reported are exactly as demanded. The syntax errors reported are in sync with the heuristic of the panic recovery mode used and all of them are detected where they are present. We’ve formally verified the results by making the parse trees on paper.

8. Compilation Details:

[A]. Makefile works (yes/no): **YES**

[B]. Code Compiles (yes/ no): **YES**

[C]. Mention the .c files that do not compile: **N/A**

[D]. Any specific function that does not compile: **N/A**

[E]. Ensured the compatibility of your code with the specified gcc version(yes/no) **YES**

9. **Driver Details**: Does it take care of the options specified earlier(yes/no): **YES**

10. **Execution**

[A]. status (describe in maximum 2 lines): We are able to obtain the desired output for parse tree through n-ary in-order traversal for syntactically correct code. Further, we are able to report syntax and/or lexical errors appropriately for the **entire** code.

[B]. Execution time taken for

- t1.txt CPU Time: 12.000000, CPU time(in seconds): 0.000012
- t2.txt CPU Time: 88.000000, CPU time(in seconds): 0.000088
- t3.txt CPU Time: 227.000000, CPU time(in seconds): 0.000227
- t4.txt CPU Time: 777.000000, CPU time(in seconds): 0.000777
- t5.txt CPU Time: 803.000000, CPU time(in seconds): 0.000803
- t6.txt CPU Time: 1087.000000, CPU time(in seconds): 0.001087

[C]. Gives segmentation fault with any of the test cases (1-6) uploaded on the course page. If yes, specify the testcase file name: N/A

11. Specify the language features your lexer or parser is not able to handle (in maximum one line): Reporting symbols such as apostrophe (‘), at (@) etc. as lexical errors (this, though, is an implementation issue, and the code can always be modified to report them as errors, if required).

12. Are you availing the lifeline (Yes/No): YES

13. Declaration: We, **AYUSH JAIN, BHARAT BHARGAVA and SATVIK GOLECHHA**, declare that we have put our genuine efforts in creating the compiler project code and have submitted the code developed only by our group. We have not copied any piece of code from any source. If our code is found plagiarized in any form or degree, we understand that a disciplinary action as per the institute rules will be taken against us and we will accept the penalty as decided by the department of Computer Science and Information Systems, BITS, Pilani. [Write your ID and names below]

ID **2017A7PS0093P** Name: **AYUSH JAIN**

ID **2017A7PS0025P** Name: **BHARAT BHARGAVA**

ID **2017A7PS0117P** Name: **SATVIK GOLECHHA**

Date: 24th Feb, 2020
