BCSE III - Compiler Design Lab

Project 10

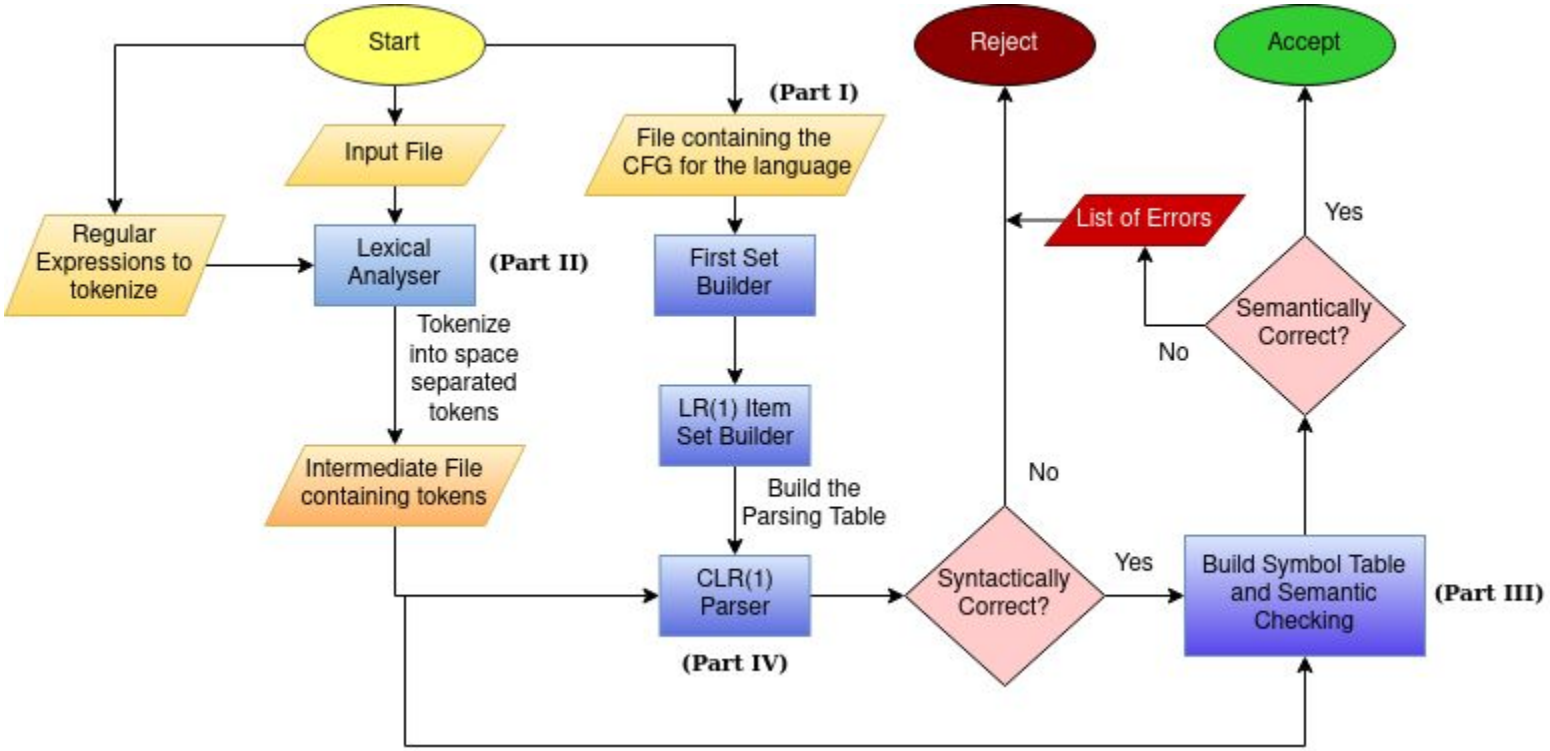
Link to Project

Problem Statement:

Consider a simple C-like language with
Data Type: integer(**int**), floating point(**float**) and **void**
Declaration statements can appear anywhere in the program and scope rules are to be maintained accordingly.
Condition constructs: **if**, **else**. (**if** without **else** and nested statements are not supported).
Assignments to the variables are performed using the input/output constructs:
read x - Read into variable x
print x - Write variable x to output.
Arithmetic operators {+, -, *, /, %, ++} are supported
Relational operators used in the **if** statements are {‹, ›, =}
There may be more than one functions in the program.

Part I - Construct a CFG for this language.
Part II - Write lexical analyser to scan the stream of characters from a program written in the above language and generate stream of tokens.
Part III - Maintain a symbol table with appropriate data structures. The symbol table should also maintain the scope of each variable declared within the program.
Part IV - Write a Canonical LR(1) parser for this language (modules include LR(1) item set construction, parsing table construction and parsing).

Flow Diagram of the Project



Part I - CFG

- Grammar supports all the features mentioned in the problem statement.
- Each program must contain a main function following zero or more numbers of other functions.
- Program can have nested if-else definitions.
- Expressions can be used in conditionals along with normal ones.
- Function calls can be used in expressions, conditions and print statements.
- Function calls can be passed as parameters to another function call.

Part I - CFG

Every grammar consists of quadruples (N,T, P, S). Our grammar specification is -

Nonterminals - prog, func, main_func, funcs, data_type, params, param, decl, stmts, stmt, norm, io, ret, exp, const, io_op, E, T, F, H, bool_op, B, if_el, func_call, args

Terminals - void, int, float, (,), {, }, main, id, $, \,, ;, +, -, *, /, %, <, >, =, read, write, return, c_int, c_float, ++, <=, >=, ==, if, else

Productions -

prog -> funcs main_func|main_func
funcs -> funcs func|func
main_func -> void main ( ) { stmts }|void main ( ) { }

func -> void id ( ) { stmts }|data_type id ( ) { stmts }|void id (params) { stmts }|data_type id (params) { stmts }|void id ( ) { }|data_type id ( ) { }|void id (params) { }|data_type id (params) { }

Part I - CFG

Productions -

data_type -> int|float
params -> params , param|param
param -> data_type id
stmts -> stmts stmt|stmt|stmts if_el|if_el
stmt -> decl|norm
decl -> data_type id ;
norm -> io ;|ret ;|exp ;

if_el -> if (B) { stmts } else { stmts }|if (B) { stmts } else { }|if (B) { } else { stmts }|if (B) { } else { }

Start Symbol - prog

Productions -

io -> io_op id|io_op const
io_op -> read|write
const -> c_int|c_float
ret -> return id|return const
exp -> id = E|E
E -> E + T|E - T|T
T -> T * F|T / F|T % F|F
F -> id ++|H
H -> id|const|func_call
B -> H bool_op H
bool_op -> <|>|==|<=|>=
func_call -> id (args)|id ( )
args -> args , H|H

Part II - Lexical Analyser

- Takes in Regular Expressions and uses them to tokenize the input file into space separated tokens.
- Converts constants into their grammar accepting terminal forms, i.e. 5 -> **c_int** and 3.14 -> **c_float**.
- Converts identifiers into terminal **id**, except keywords, which are also terminal themselves.

Input File:

Project > my_inp.txt

```
1   int sq(int a) {
2       int b;
3       b=a*a;
4       return b;
5   }
6   void main() {
7       int x;
8       read x;
9       if (sq(x) > 5) {
10          write 3.14;
11      } else {
12          write x;
13      }
14  }
```

Intermediate File containing tokens:

intermediate_file.txt

Project > intermediate_file.txt

```
1   int id ( int id ) { int id ; id = id * id ; return id ; } void main ( ) { int id ; read id ; if ( id ( id ) > c_int ) { write c_float ; } else { write id ; } }
```

Part III - Symbol Table

- Designed Symbol Table using Multiple Hash Tables, implemented as a list of hash tables.
- One hash table for each currently visible scope.
- Whenever a token is inserted in the symbol table, the lexeme value of the **id** is the key, and the value consists of attributes like a flag to determine if it's a variable or a function, data type(return type for functions), and parameter list for functions.
- During scanning, whenever it enters a new block of code, a new hash table is generated in the head of the list, and when it exits the block, the hash table in the head of the list is deleted.

Symbol Table for Previous Input File:

Project > symtab.txt

```
1
2   --------Exiting a block: --------
3   Symbol Table for this block was:
4   Id : b -> Data Type : int -> Type : Variable
5   Id : a -> Data Type : int -> Type : Variable
6
7   --------Exiting a block: --------
8   Symbol Table for this block was:
9
10  --------Exiting a block: --------
11  Symbol Table for this block was:
12
13  --------Exiting a block: --------
14  Symbol Table for this block was:
15  Id : x -> Data Type : int -> Type : Variable
16
17  --------Exiting a block: --------
18  Symbol Table for this block was:
19  Id : main -> Data Type : void -> Type : Function -> Parameters :
20  Id : sq -> Data Type : int -> Type : Function -> Parameters : <int, a>,
21
```

Part IV - CLR(1) Parser

- Parser will take the grammar (written in a pre-defined format) as input at the beginning.
- Then it will build the **First Set** of each terminals and non-terminals.
- With the help of the **First Set** it will build the **Item Set** next.
- From the **Item Set** it will construct the **Parsing Table**.
- With the help of **Parsing Table** it will parse any input **stream of tokens** tokenized by the Lexical Analyser, given as input.

CFG:

```
1   #
2   S|A|B
3   #
4   a|b
5   #
6   S@A B
7   A@A a|a
8   B@B b|b
9   #
10  S
11  #
```

First Set:

```
1   First Set of terminals and non-terminals------
2   B : b,
3   S : a,
4   A : a,
5   $ : $,
6   ε : ε,
7   b : b,
8   a : a,
9
```

Part IV - CLR(1) Parser

Item Set:

```
Item No. : 0 -----
S' -> . S   lookahead : $
S -> . A B  lookahead : $
A -> . A a  lookahead : a b
A -> . a    lookahead : a b

Item No. : 1 -----
S' -> S     lookahead : $

Item No. : 2 -----
S -> A . B  lookahead : $
A -> A . a  lookahead : a b
B -> . B b  lookahead : b $
B -> . b    lookahead : b $

Item No. : 3 -----
A -> a  lookahead : a b

Item No. : 4 -----
S -> A B    lookahead : $
B -> B . b  lookahead : b $

Item No. : 5 -----
A -> A a    lookahead : a b

Item No. : 6 -----
B -> b  lookahead : b $

Item No. : 7 -----
B -> B b    lookahead : b $

Edge List -----
    From    To  On symbol
    0   1   S
    0   2   A
    0   3   a
    2   4   B
    2   5   a
    2   6   b
    4   7   b
```

Parsing Table:

```
Table entry for Item 0 -----
GOTO Entries
On symbol : a Op name : shift goto : 3
ACTION Entries
On symbol : S Op name : goto goto : 1
On symbol : A Op name : goto goto : 2


Table entry for Item 1 -----
GOTO Entries
ACTION Entries


Table entry for Item 2 -----
GOTO Entries
On symbol : a Op name : shift goto : 5
On symbol : b Op name : shift goto : 6
ACTION Entries
On symbol : B Op name : goto goto : 4


Table entry for Item 3 -----
GOTO Entries
On symbol : a Op name : reduce Reduce : A -> a
On symbol : b Op name : reduce Reduce : A -> a
ACTION Entries


Table entry for Item 4 -----
GOTO Entries
On symbol : b Op name : shift goto : 7
ACTION Entries


Table entry for Item 5 -----
GOTO Entries
On symbol : a Op name : reduce Reduce : A -> A a
On symbol : b Op name : reduce Reduce : A -> A a
ACTION Entries


Table entry for Item 6 -----
GOTO Entries
On symbol : b Op name : reduce Reduce : B -> b
ACTION Entries


Table entry for Item 7 -----
GOTO Entries
On symbol : b Op name : reduce Reduce : B -> B b
ACTION Entries
```

Semantic Checks

- Some of the basic semantic checks are performed too using the Symbol Table.
- Whenever a variable is declared or used in an expression, scope check is performed to prevent multiple declarations and to check existence of variable respectively.
- Also, since multiple functions can be present in our language. Whenever a certain function is called, the number of parameters passed is also checked. The parameters can be either identifiers, constants or even other function calls.

Input File:

Project > my_inp.txt

```
1   int sq(int a) {
2       int b;
3       b=a*a;
4       return b;
5   }
6   void main() {
7       int x;
8       x=sq(sq(5));
9       read y;
10      if (sq() > 5) {
11          int x;
12          read x;
13          int x;
14          write 3.14;
15      } else {
16          write x;
17      }
18  }
```

Semantic Errors:

```
Undeclared identifier y used.
Incorrect call of function sq .
Multiple Declarations. x is already declared.
Finished Semantic Analysis.
Input your choice-----
 1. Verify a input file
 2. Exit

Enter your choice : 2
kdjonty@KDJonty-Ubuntu-20:~/CSE/Sem6/Compiler/Lab/Project$
```

Thanks!

Our Team:

- Koustav Dhar - Roll: 001910501022
- Soumitra Sinhahajari - Roll: 001910501012
- Tasmiul Alam Shopnil - Roll: 001910501004
- Wreetbhas Pal - Roll: 001910501011
- Moinuddin Sheikh - Roll: 401910501002