Team

Bishoy Nader Fathy Beshara	22
Amr Mohamed Nasr Eldine Elsayed	45
Marc Magdi Kamel Ayoub	52
Michael Raafat Mikhail Zaki	54

Compiler Phase II : Parser Generator

28th March 2018

USED DATA STRUCTURES

- **We defined the following Model Classes:**
    1. GrammarElement: represents the class of representing the terminal in grammar rule, it contains its name only.
    2. NonTerminal: is the child of GrammarElement and has extra pointers to the set of expressions it leads to, a set of strings that represents its first set, a set of strings that represents its follow set, pointers to the expressions that reference it, and a boolean epsilon that signifies if it evaluates to epsilon .
    3. GrammarExpression: represent a term in the RHS in a grammar rule, it has pointer to the LHS NonTerminal and a vector of pointers the grammar elements and non-terminals that constitute the RHS, a set of strings that represents its first set and a boolean epsilon that signifies if the set has in its first set an epsilon(If the expression evaluates to epsilon).
    4. GrammarTable: data structure that contains the grammar table data, it's used in the syntactical generator, and analyzer. This table is saved and load in two formats, one for the parser and one in human readable format.
    5. FirstElementWrapper: data structure that acts as a temporary wrapper for the non-terminals when calculating their first set. It contains sets to both the strings in the first set, and the other non-terminals whose first strings should also be included.
    6. FirstExpressionWrapper: data structure that acts as a temporary wrapper for the grammar expressions when calculating their first set. It contains sets to both the

strings in the first set, and the other non-terminals whose first strings should also be included.

7. FollowElementWrapper :data structure that acts as a temporary wrapper for the non-terminals when calculating their follow set. It contains sets to both the strings in the follow set, and the other non-terminals whose follow strings should also be included.

- **We defined enums in grammar table parser to help on handling the splitted rules among different lines.**
- **We used the following data structures:**
    1. Stack: Used in parsing input in the syntactical analyzer to match the input token with the grammar table.
    2. Map: As the data structure to hold the grammar table, the key is a pair of strings and the value is a vector of strings
    3. Vector: To hold any related data, ex: in the grammar table hold the derivations.
    4. Set: To hold the synch values in the grammar table, also to hold any related data such as all rules or all expressions to be more easy when freeing the memory.
    5. Unordered maps were used in follow and first calculators to map to non-terminal its temporary wrapper, and in first calculator to also map each grammar expression to its wrapper, alongside vectors that were used to keep reference to all the wrappers so we can free their memory easily after usage.
    6. Unordered sets were used to hold the strings of the first or follow sets of expressions or elements since order was not significant .

ALGORITHMS AND TECHNIQUES USED

1. REGEX for grammar parsing

We used regex to check the validity of the grammar file and extract the values of Non-Terminals and Terminals to build owr model.

2. First and Follow Sets Calculation

We used the simple rules of first and follow to calculate them for each grammar element before left recursion and after left recursion and a third time after left factoring for the possibility of addition of new non terminals in eliminating left factoring or recursion.
To calculate first, we simply added suitable terminals to the set of strings of the first set to each suitable expression/element, and in case of a non-terminal we added a pointer to that non-terminal as a member of the first, and we checked if a non-terminal is eps then

repeat this operation on the following symbol. Then we merge the first set of strings of a non-terminal that is a member of the first set of another to the others's set. An operation that is similar to the union of two sets. We repeat this operation until no sets are changed. Follow calculation follows the same logic.

3. Elimination of left recursion

We used algorithm discussed in lecture to eliminate left recursion after we check the presence of left recursion.

4. Building grammar table

We start by looping through the non terminals, for every non terminal we go through the list of expressions and get the first for every expression, and then add a new element to the grammar table with this expression. if expression can lead to eps, add eps to all follows of grammarElement. if current element leads to eps, add eps o/p to follow, if not add synch to follows.

5. Parsing input tokens

We used the stack data structure to test the input with the grammar table built, we also made a panic mode recovery routine to check for errors during parsing and printing the suitable error message.

6. Unit Tests

We wrote extensive unit tests for each module in the project and mocked the input and output of this module also wrote code to compare both expected and actual outputs to make code debugging neat and easy,

Integration To Phase 1

We integrated our project to phase 1 seamlessly because of our decision to separate our top level tokenizer from the lexical analyzer main, we could use it beside our new models by just calling hasToken() and nextToken() and passing them to the parser which took one token at a time.

Our usage to intermediate grammar and transition table means that our syntactic analyzer needs to only read these files not generate them. Since we are convinced that a generator should be ran one time for each rule file, but the analyzer can run multiple times, we also did this in phase 2.

So a user should give the lexical rules file to the lexical generator to get a transition table file. Then give the grammar rules file to the syntactic generator to get a grammar table file. The analyzer will then just ask him to give him the transition table file and the grammar table file and the input file to generate the required output.

INPUT FILES

Lexical Rules Input File

```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digits ( \L | E digits)
relop: \=\= | !\= | > | >\= | < | <\=
assign: \=
{ if else while }
[; , \( \) { }]
addop: \+ | \-
mulop: \* | /
```

Grammar Rules Input File

```
# METHOD_BODY ::= STATEMENT_LIST
# STATEMENT_LIST ::= STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT ::= DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION ::= PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE ::= 'int' | 'float'
# IF ::= 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
# WHILE ::= 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
# ASSIGNMENT ::= 'id' 'assign' EXPRESSION ';'
```

EXPRESSION ::= SIMPLE_EXPRESSION
| SIMPLE_EXPRESSION 'relop' SIMPLE_EXPRESSION
SIMPLE_EXPRESSION ::= TERM | SIGN TERM | SIMPLE_EXPRESSION 'addop' TERM
TERM ::= FACTOR | TERM 'mulop' FACTOR
FACTOR ::= 'id' | 'num' | '(' EXPRESSION ')'
SIGN ::= 'addop' | 'addop'

AFTER ELIMINATING LEFT RECURSION AND FACTORING

METHOD_BODY -> STATEMENT_LIST
STATEMENT_LIST -> STATEMENT STATEMENT_LIST'
STATEMENT_LIST' -> STATEMENT STATEMENT_LIST' | EPS
STATEMENT -> DECLARATION | IF | WHILE | ASSIGNMENT
DECLARATION -> PRIMITIVE_TYPE 'id' ';'
PRIMITIVE_TYPE -> 'int' | 'float'
IF -> 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
WHILE -> 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
ASSIGNMENT -> 'id' 'assign' EXPRESSION ';'
EXPRESSION -> SIMPLE_EXPRESSION EXPRESSION_1
EXPRESSION_1 -> 'relop' SIMPLE_EXPRESSION | EPS
SIMPLE_EXPRESSION -> TERM SIMPLE_EXPRESSION' | SIGN TERM SIMPLE_EXPRESSION'
SIMPLE_EXPRESSION' -> 'addop' TERM SIMPLE_EXPRESSION' | EPS
TERM -> FACTOR TERM'
TERM' -> 'mulop' FACTOR TERM' | EPS
FACTOR -> 'id' | 'num' | '(' EXPRESSION ')'
SIGN -> '+' | '-'

THE RESULTANT TRANSITION TABLE FOR THE MINIMAL DFA

Fully spreadsheet is found here for the resultant transition table:

https://docs.google.com/spreadsheets/d/1l55ldCE-sXushCMeCztCe_kkqaiwBOaBKNgSWLk_6a0/edit?usp=sharing

THE RESULTANT Grammar TABLE

Fully spreadsheet is found here for the resultant grammar table:

https://docs.google.com/spreadsheets/d/1Tk03EgcoMznCsVab8LEBp7wlrDDrplU8B25rpam1oLo/edit?usp=sharing

TEST PROGRAM

```
int x;
x = 5;
if (x > 2)
{
x = 0;
} else {
x = 10;
}
```

Analyzer Output:

METHOD_BODY
STATEMENT_LIST
STATEMENT STATEMENT_LIST'
DECLARATION STATEMENT_LIST'
PRIMITIVE_TYPE id ; STATEMENT_LIST'
int id ; STATEMENT_LIST'
int id ; STATEMENT STATEMENT_LIST'
int id ; ASSIGNMENT STATEMENT_LIST'
int id ; id assign EXPRESSION ; STATEMENT_LIST'

int id ; id assign SIMPLE_EXPRESSION EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign TERM SIMPLE_EXPRESSION' EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign num SIMPLE_EXPRESSION' EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign num EXPRESSION_1 ; STATEMENT_LIST'
int id ; id assign num ; STATEMENT_LIST'
int id ; id assign num ; STATEMENT STATEMENT_LIST'
int id ; id assign num ; IF STATEMENT_LIST'
int id ; id assign num ; if (EXPRESSION) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (SIMPLE_EXPRESSION EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (TERM SIMPLE_EXPRESSION' EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id TERM' SIMPLE_EXPRESSION' EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id SIMPLE_EXPRESSION' EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id EXPRESSION_1) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id relop SIMPLE_EXPRESSION) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id relop TERM SIMPLE_EXPRESSION') { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id relop FACTOR TERM' SIMPLE_EXPRESSION') { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id relop num TERM' SIMPLE_EXPRESSION') { STATEMENT } else { STATEMENT } STATEMENT_LIST'
int id ; id assign num ; if (id relop num SIMPLE_EXPRESSION') { STATEMENT } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { STATEMENT } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { ASSIGNMENT } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign SIMPLE_EXPRESSION EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign TERM SIMPLE_EXPRESSION' EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num SIMPLE_EXPRESSION' EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num EXPRESSION_1 ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { STATEMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { ASSIGNMENT } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign EXPRESSION ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign SIMPLE_EXPRESSION EXPRESSION_1 ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign TERM SIMPLE_EXPRESSION' EXPRESSION_1 ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION_1 ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign num SIMPLE_EXPRESSION' EXPRESSION_1 ; } STATEMENT_LIST'

int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign num EXPRESSION_1 ; } STATEMENT_LIST'
int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign num ; } STATEMENT_LIST'
int id ; id assign num ; if (id relop num) { id assign num ; } else { id assign num ; }

ASSUMPTIONS AND JUSTIFICATIONS

- Grammar rules are written LHS ::= RHS.
- EPS is a terminal and is represented as '\L' in the grammar rules file.
- We add (_i) to new productions that are made from left factoring of the original one, where i is the number of factors in the original production.
- We add (') to new productions that are made from left recursion of the original one
- Reserved characters are ', |, :, =, # and when need to use them in names they must be preceded by \.

BONUS

1. Elimination of Left recursion:

We applied the algorithm discussed in lecture, we check first for the presence of direct recursion in the grammar rule, If yes we eliminate it be removing first element in the expressions which cause this left recursion and form a new rule and put this expressions in, all expressions we add the new rule at the end of each expression, If not direct Then we check for indirect left recursion and put all expression that lead to indirect left recursion in a list, then we substitute them to form a direct left recursion rule and solve it as we mentioned before.

2. Left Factoring:

We first check for a direct left factoring and start to have a factor of elements then we put them in an expression and add a new grammar element to this expression, then we take the expression involved in this factorization and put them in this new grammar element, we also check the nested left factoring then by substituting the same as we have done in left recursion we get a direct left factoring then apply the same technique of direct left factoring.