Compilers

Phase 2

Parser Generator

Febronia Ashraf Tawfik (31)

MennaTallah Mohamed Ahmed Osman (55)

Merna Mustafa El-Refaie (57)

Neveen Samir Nagy (61)

A description of used data structures

- ***Struct***

  struct transition → is used for transition diagrams to build transitions between states.

  struct Non_terminal → is used for nonterminal symbols as each symbol has a name and vector of productions.

- ***Vector***

  vector<Non_terminal> all_nonTerminals → is used to store all nonterminals.

  vector<string> terminals → is used to store terminals.

  vector<pair<string,Transition_Digrams>> tran_Digram → is used to store transitions between states for transition diagrams for grammar.

- ***Map***

  map<string, set<string> > first → is used to store each nonterminal with its first sets as the key of the map is the name of the nonterminal and the value is the set.

  map<string, set<string> > follow → is used to store each nonterminal with it's follow sets as the key of the map is the name of the nonterminal and the value is the set.

  map<string, set<pairs> > firstForTable → is used to store each nonterminal with its first sets as the key of the map is the name of the nonterminal and the value is pairs which hold the set of first for nonterminal and the corresponding production to each terminal in first set.

  map<string,vector<string> > terminalsColumn →

- ***Set***

  set<string> → is used to store the first and follow sets of each nonterminal.

All algorithms and techniques used

- **Void Read_inputFile(file_name):**

  First this function is used to read the file of the grammar and constructs the vectors of the terminals and nonterminal symbols, constructs the productions of each nonterminal.

- **Vector <non_terminal> left_factoring(vector<non_terminal>):**

  After reading the input file of grammar this function is used to check if any nonterminal has factoring.

- **Void left_recurion(vector<non_terminal>):**

  This function is used after that to remove any left recursion found in any nonterminal.

- **Void construct_transitionDigram():**

  After calling the above functions, all nonterminal symbols have no factoring or recursion, so this function is called to build the transition diagram for the grammar.

- **Void set_first():**

  Then This function is called to get the first set for each nonterminal symbol.

- **Void set_follow():**

  Then This function is called to get the follow set for each nonterminal symbol.

- **map<string, map<string, vector<string>>> build_table():**

  This function is used to build the parsing table by using the first & follow sets of each nonterminal.

- **Void leftMostDerivation():**

  Finally, this function is called to check the input test file, if it is accepted or has error(s).

Transition Diagrams

```
Name: METHOD_BODY
q0 --> q1 : Symbol is STATEMENT_LIST

The final state is q1

Name: STATEMENT_LIST
q2 --> q3 : Symbol is STATEMENT
q3 --> q4 : Symbol is STATEMENT_LIST_dash

The final state is q4

Name: STATEMENT_LIST_dash
q5 --> q6 : Symbol is STATEMENT
q6 --> q7 : Symbol is STATEMENT_LIST_dash
q5 --> q7 : Symbol is ^

The final state is q7

Name: STATEMENT
q8 --> q9 : Symbol is DECLARATION
q8 --> q9 : Symbol is IF
q8 --> q9 : Symbol is WHILE
q8 --> q9 : Symbol is ASSIGNMENT

The final state is q9

Name: DECLARATION
q10 --> q11 : Symbol is PRIMITIVE_TYPE
q11 --> q12 : Symbol is id
q12 --> q13 : Symbol is ;

The final state is q13

Name: PRIMITIVE_TYPE
q14 --> q15 : Symbol is int
q14 --> q15 : Symbol is float

The final state is q15
```

```
Name: IF
q16 --> q17 : Symbol is if
q17 --> q18 : Symbol is (
q18 --> q19 : Symbol is EXPRESSION
q19 --> q20 : Symbol is )
q20 --> q21 : Symbol is {
q21 --> q22 : Symbol is STATEMENT
q22 --> q23 : Symbol is }
q23 --> q24 : Symbol is else
q24 --> q25 : Symbol is {
q25 --> q26 : Symbol is STATEMENT
q26 --> q27 : Symbol is }

The final state is q27

Name: WHILE
q28 --> q29 : Symbol is while
q29 --> q30 : Symbol is (
q30 --> q31 : Symbol is EXPRESSION
q31 --> q32 : Symbol is )
q32 --> q33 : Symbol is {
q33 --> q34 : Symbol is STATEMENT
q34 --> q35 : Symbol is }

The final state is q35

Name: ASSIGNMENT
q36 --> q37 : Symbol is id
q37 --> q38 : Symbol is =
q38 --> q39 : Symbol is EXPRESSION
q39 --> q40 : Symbol is ;

The final state is q40

Name: EXPRESSION
q41 --> q42 : Symbol is SIMPLE_EXPRESSION
q42 --> q43 : Symbol is EXPRESSION`

The final state is q43

Name: SIMPLE_EXPRESSION
q44 --> q45 : Symbol is TERM
q45 --> q47 : Symbol is SIMPLE_EXPRESSION_dash
q44 --> q45 : Symbol is SIGN
q45 --> q46 : Symbol is TERM
q46 --> q47 : Symbol is SIMPLE_EXPRESSION_dash

The final state is q47
```

```
Name: SIMPLE_EXPRESSION_dash
q48 --> q49 : Symbol is addop
q49 --> q50 : Symbol is TERM
q50 --> q51 : Symbol is SIMPLE_EXPRESSION_dash
q48 --> q51 : Symbol is ^

The final state is q51

Name: TERM
q52 --> q53 : Symbol is FACTOR
q53 --> q54 : Symbol is TERM_dash

The final state is q54

Name: TERM_dash
q55 --> q56 : Symbol is mulop
q56 --> q57 : Symbol is FACTOR
q57 --> q58 : Symbol is TERM_dash
q55 --> q58 : Symbol is ^

The final state is q58

Name: FACTOR
q59 --> q62 : Symbol is id
q59 --> q62 : Symbol is num
q59 --> q60 : Symbol is (
q60 --> q61 : Symbol is EXPRESSION
q61 --> q62 : Symbol is )

The final state is q62

Name: SIGN
q63 --> q64 : Symbol is +
q63 --> q64 : Symbol is -

The final state is q64

Name: EXPRESSION`
q65 --> q67 : Symbol is ^
q65 --> q66 : Symbol is relop
q66 --> q67 : Symbol is SIMPLE_EXPRESSION

The final state is q67
```

Parsing Tables

Because terminals and non terminals may be quite big. Predictive parsing table could be represented like this

- Non_terminal --->
- if terminal then value
- if terminal then value
- etc...

Which we could consider non terminal as column, terminal as row and value as table cell

For example:
for the following 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' '\=' 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 = '+' | '-'
```

After transforming the grammar to be a LL(1) grammar by left factoring the grammar and eliminating left recursion from it.

Predictive parsing table is :

```
*****************************************************************************************************
The predictive parsing table :
*****************************************************************************************************
Non terminal : ASSIGNMENT -->
If $ then synch
If float then synch
If id then id = EXPRESSION ;
If if then synch
If int then synch
If while then synch
If } then synch

-----------------------------------------------------------------------------------------------------
Non terminal : DECLARATION -->
If $ then synch
If float then PRIMITIVE_TYPE id ;
If id then synch
If if then synch
If int then PRIMITIVE_TYPE id ;
If while then synch
If } then synch

-----------------------------------------------------------------------------------------------------
Non terminal : EXPRESSION -->
If ( then SIMPLE_EXPRESSION EXPRESSION`
If ) then synch
If + then SIMPLE_EXPRESSION EXPRESSION`
If - then SIMPLE_EXPRESSION EXPRESSION`
If ; then synch
If id then SIMPLE_EXPRESSION EXPRESSION`
If num then SIMPLE_EXPRESSION EXPRESSION`

-----------------------------------------------------------------------------------------------------
Non terminal : EXPRESSION` -->
If ) then ^
If ; then ^
If relop then relop SIMPLE_EXPRESSION

-----------------------------------------------------------------------------------------------------

Non terminal : FACTOR -->
If ( then ( EXPRESSION )
If ) then synch
If ; then synch
If addop then synch
If id then id
If mulop then synch
If num then num
If relop then synch

-----------------------------------------------------------------------------------------------------
Non terminal : IF -->
If $ then synch
If float then synch
If id then synch
If if then if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
If int then synch
If while then synch
If } then synch

-----------------------------------------------------------------------------------------------------
Non terminal : METHOD_BODY -->
If $ then synch
If float then STATEMENT_LIST
If id then STATEMENT_LIST
If if then STATEMENT_LIST
If int then STATEMENT_LIST
If while then STATEMENT_LIST

-----------------------------------------------------------------------------------------------------
Non terminal : PRIMITIVE_TYPE -->
If float then float
If id then synch
If int then int

-----------------------------------------------------------------------------------------------------
Non terminal : SIGN -->
If ( then synch
If + then +
If - then -
If id then synch
If num then synch
```

```
-----------------------------------------------------------------------------------------------
Non terminal : SIMPLE_EXPRESSION -->
If ( then TERM SIMPLE_EXPRESSION_dash
If ) then synch
If + then SIGN TERM SIMPLE_EXPRESSION_dash
If - then SIGN TERM SIMPLE_EXPRESSION_dash
If ; then synch
If id then TERM SIMPLE_EXPRESSION_dash
If num then TERM SIMPLE_EXPRESSION_dash
If relop then synch

-----------------------------------------------------------------------------------------------
Non terminal : SIMPLE_EXPRESSION_dash -->
If ) then ^
If ; then ^
If addop then addop TERM SIMPLE_EXPRESSION_dash
If relop then ^

-----------------------------------------------------------------------------------------------
Non terminal : STATEMENT -->
If $ then synch
If float then DECLARATION
If id then ASSIGNMENT
If if then IF
If int then DECLARATION
If while then WHILE
If } then synch

-----------------------------------------------------------------------------------------------
Non terminal : STATEMENT_LIST -->
If $ then synch
If float then STATEMENT STATEMENT_LIST_dash
If id then STATEMENT STATEMENT_LIST_dash
If if then STATEMENT STATEMENT_LIST_dash
If int then STATEMENT STATEMENT_LIST_dash
If while then STATEMENT STATEMENT_LIST_dash

-----------------------------------------------------------------------------------------------

Non terminal : STATEMENT_LIST_dash -->
If $ then ^
If float then STATEMENT STATEMENT_LIST_dash
If id then STATEMENT STATEMENT_LIST_dash
If if then STATEMENT STATEMENT_LIST_dash
If int then STATEMENT STATEMENT_LIST_dash
If while then STATEMENT STATEMENT_LIST_dash

-----------------------------------------------------------------------------------------------
Non terminal : TERM -->
If ( then FACTOR TERM_dash
If ) then synch
If ; then synch
If addop then synch
If id then FACTOR TERM_dash
If num then FACTOR TERM_dash
If relop then synch

-----------------------------------------------------------------------------------------------
Non terminal : TERM_dash -->
If ) then ^
If ; then ^
If addop then ^
If mulop then mulop FACTOR TERM_dash
If relop then ^

-----------------------------------------------------------------------------------------------
Non terminal : WHILE -->
If $ then synch
If float then synch
If id then synch
If if then synch
If int then synch
If while then while ( EXPRESSION ) { STATEMENT }
If } then synch

-----------------------------------------------------------------------------------------------
```

Comments about used tools

Explanation of functions of all phases

Phase 1:

- **Class FA**→ that is used to construct the finite automata, the deterministic and the nondeterministic, it holds the number of the states in the FA, the start state, final state and a vector of the final state if the finite automata has multiple final states, the functions that are used in this class are set and get calls as to set the number of the states and get it if is needed, one of its functions is set transition which set all the transitions in the finite automata in vector of struct.
- **Function concatenation** → that takes two FA (class) and concatenate them by build another FA that is the result, its number of the states is the sum of the number of states in the two FA and its start state is the start state of one of the two FA and its final state is the final state of the second FA and the final state of the first FA will be the start state of the second FA.
- **Function kleene closure** → it takes FA as input and build the result FA which their number of states equal the number of the states of input FA plus two, the start state of the result goes to start state of the input FA with absalon transition and the same for final state as the final state of the input FA goes to final state of the result with absalon transition and the final state of the input FA goes to start state of the input FA by absalon transition and the same with start state of the result as goes to final state of the result by absalon transition.
- **Function positive closure** → the same as kleene closure but without the transition between start state and final state of the result FA.
- **Function union** → as takes a vector of FAs that will be union together, build a result FA that has a number of states equal the sum of all number of states of input FAs plus 2 as the start state of result FA will go to all start states of the input FAs by absalon transition and all the final states of the input FAs will go to result's final state by absalon transition.

- **Function keywords** → it builds the NFA of the input keywords, it takes a string as input which consists of keywords and builds the NFA to them, then puts the result in its map.
- Function punctuations → it takes a string as input which consists of symbols and builds the NFAs to them then puts the result in its map.
- **Function return FA** → takes string as input and checks if it has a NFA built before, if exists then returns its NFA if not then builds its NFA and returns it.
- **Function range** → when the input expression has (-) operator which is considered a range between two values so, this function takes all possible values between these two values and builds the FA for all these values then union them together.
- **Function expression to NFA** → it is used to build the FA for regular expressions and regular definitions, it takes string -the name of the regular- and its expression, a for loop is used to check every character in expression string, if it is a character then it is kept in a string, if it is a space then calculate the NFA for the string before it, if it is a special character then it checked, if it is (\) operators then takes the next character whatever it is type, if it is (or(|) range(-) (() concatenation) then push it to operators stack, if it is ()) operator then execute all the operations that are inside this operators until we arrive to (operators, after that if there were operators that haven't been executed yet, then it is executed then push the final NFA to its corresponding map depending on the third input string that determines the type of the input string.
- **Function read file** → reads the input file and call functions keywords, punctuation, expression to NFA depending on input line in the file.
- **Function language** → it combines all NFAs of regular expressions, keywords and punctuations in one map then calls function union NFA that union them.
- **Function union NFA** → it takes a vector of all NFA that are built during reading the input file and union it by make the result FA's start state goes to all start state of input FAs by absalon transition, and result FA has a vector of final states that are all final states of the input FAs.
- **Function getNonFinalStates**→ it takes a vector of all vertices as first parameter and another vector of final states as second parameter and just take the difference of both to get non final states
- **Function getInputs**→ it takes one parameter which is all the transitions of DFA and gets all inputs that make these transitions.

- **Function NFAtoDFA** → it takes NFA and transforms it to DFA. It gets s0 closure and makes it the start state then gets the next state and its equivalent for every input symbol. Before adding any state to the table it checks if this state is added before or not and then checks if it's a final state or not.
- **Function findNextStats**→ it takes four parameters ,integer represent current vertex, vector of all the transitions of DFA, vector of all the inputs and map contains vertex as key and its partition as value.For each input we check every transition if source (vertex_from) of the transition equals the current vertex and input that make a transition equals input of vector then we store the destination (vertex_to) in string which is the output of the function.
- **Function updateMapValues**→ it takes three parameters integer which represents the new value of partition name , vector contains vertices of the new partition and map contains vertex as key and its partition as value . For each vertex in the vector we change the value of the map with key equals that vertex.
- **Function minimization**→ it takes one parameter class DFA and use the last four Function above
    1. it uses updateMapValues to identify the final and non final states partition name then store them in a map called keyOfStates which its key is vertex and its value is partition name
    2. Add the final and non final states in another map called partitions which its key is partition name and its value is vector of all vertices in each the vector with key equal zero used to know which partitions must be split.
    3. Another vector called mySS contains partition names that must be checked for minimization.
    4. For each vertex in vector currntVector - at first contains non final states- we get its next states using findNextStats function and use the output string as key in pre_result map this allow us to know which vertices have the same next states for minimization because all vertices with the same next states are in the vector represents the value of pre_result map with next states as key
    5. When we check all vertices in currntVector if pre_result have more than one elements that means that currntVector must be split because it has states with different next states

6. Deleting currntVector name from mySS and check if it is not empty ,change currntVector to the next partition and return to point 4
7. If mySS is empty and partition vector value for key zero is not empty ,delete all partitions with names exist in the vector of key zero and use updateMapValues function to change vertices partition names to the new names of new partitions the put all partition names again in mySS finally ,change currntVector to the next partition and return to point 4
8. if mySS and partition vector value for key zero are empty stop looping and return partition map as output for the function

- **Function minimizedTable**→ it takes output of minimization function and DFA class. this function makes class with new vertices , transitions and final states after minimization and partitioning
- **Function read test Program**→ it is used to read the test program, as it divides it into its words then sends this word to another function to check if it is included in language or not and convert it into its corresponding regular expression, keywords or punctuation.
- **Function construct output**→ it is called by function read test program. it takes two inputs a word and the minimal finite automata. It walked through the transitions of the FA and checked every character of this word with the symbol of the transitions. Finally, it checks if the states we hold are final and using priority rule, it chooses the corresponding word.

Phase 2:

- ***void read_inputFile*** → is used to read the input file which contains grammar and fills the two vectors of nonterminals and terminals.
- ***void left_recurion*** → is used to remove left recursion from grammar as it checks for every nonterminal, if it has a production that start with the name of the nonterminal which means that this nonterminal has left recursion so a new nonterminal symbol is created and is added at the end of all other productions of this nonterminal, the new nonterminal has two production, one of them is which contains the production that is founded started with the name of the nonterminal that has left recursion, this name is removed from the production and the production is added with the name of the new nonterminal at the end, the second production is epsilon.

- ***vector<Non_terminal> left_factoring*** → is used to transform the grammar into left-factored grammar which is an equivalent grammar suitable for predictive parsing, and this is implemented by iterating over the production rules and doing the following :

  For each non-terminal with two or more alternatives with a common non-empty prefix , for example , A -> b c | b d :

  First we store the common prefix (b) in a vector <string> common. Then, we modify the production to be (A -> b A` ), and store this modified production into a map<string , set< vector<string> > > modified. Finally, we make a new production ( A` -> c | d), and store it into a map<string , set< vector<string> > > dash_prods.

  After finishing the loop we gather the input productions, modified productions and the new productions (dash productions) into a vector<Non_terminal> leftFactoring, which is the new equivalent grammar, and return it.

- ***class Transition_Digrams*** → is used to build transitions for transition diagrams of grammar.
- ***void construct_transitionDigram*** → is used to build transition diagrams for each nonterminal of the grammar.
- ***void print_transitionDigram*** → is used to print the transition diagram.
- ***bool is_terminal*** → it checks if the input string is considered as a terminal symbol and returns true if string is terminal and false otherwise.
- ***void get_first*** → is used to build the first set of each nonterminal, it checks their productions if the production starts with terminal symbol (calls is_terminal) so this terminal is added to the first set of nonterminal, if starts with nonterminal, this function is called itself as recursion to get first set of this nonterminal then added its first set to the old nonterminal that has a production starts with this nonterminal.
- ***void set_first*** → is used to build the first set of each nonterminal and calls the function get_first.
- ***void get_follow*** → is used to build the follow set of each nonterminal as it walks on each production to each nonterminal to search for specific nonterminal, if found so we checks the following symbol for it, if the following is terminal (calls is_terminal) so this terminal is added to the follow set of this nonterminal, if the following is nonterminal so the first

set of it are added to the follow set of this nonterminal, if the following nonterminal has epsilon in the first set so we check for the following of the following, if there is no following, the follow set of the nonterminal of this production are added to the follow set to this nonterminal.

- ***void set_follow*** → is used to build the follow set of each nonterminal and calls the function get_follow.
- ***map<string, map<string, vector<string>>> build_table*** → is used to build the predictive parsing table. It takes the first and follow sets of each nonterminal to construct the table.
- ***queue<string> readOutAsIn*** → in this Function we get the output of the First phase and read it here.
- ***Void leftMostDerivation*** → it takes build_table as parameter which is a predictive parsing table. What it actually does is Tracing of moves made by predictive parser for certain Input and return them as output

Any assumptions made and their justification

- We assume that the epsilon symbol is '^'.
- We assume that the input file contains spaces between each two words or word and symbol.