Alexandria University,

Faculty of Engineering,

Computer and Systems Engineering Dept.

Programming Languages & Compilers

Project Report

Names:

1. ElSayed AbdelNasser ElSayed (14)
2. Ziad Hisham Ali (21)
3. Karim Atef Ahmed (33)
4. Youssef Ahmed Sayed (64)

Team Number: 3

Abstract and Objective

This project's objective is to develop a suitable Syntax Directed Translation Scheme to convert Java code to Java bytecode, performing necessary lexical, syntax and static semantic analysis (such as type checking and Expressions Evaluation.

It's divided into 3 phases:

Phase 1

This phase aims to design and implement a lexical analyzer generator tool that is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens.

Phase 2

This phase aims to design and implement an LL (1) parser generator tool that expects an LL (1) grammar as input and drive a predictive top-down parser

Phase 3

This phase aims to practice techniques of constructing semantics rules to generate intermediate code. It generates bytecode which follows the standard bytecode instructions defined in Java Virtual Machine Specification.

Introduction

Phase 1

This phase aims to design and implement a lexical analyzer generator tool that is required to automatically construct a lexical analyzer from a regular expression description of a set of tokens. The tool is required to construct a nondeterministic finite automata (NFA) for the given regular expressions, combine these NFAs together with a new starting state, convert the resulting NFA to a DFA, minimize it and emit the transition table for the reduced DFA together with a lexical analyzer program that simulates the resulting DFA machine.

Phase 2

This phase aims to design and implement an LL (1) parser generator tool that expects an LL (1) grammar as input then computes First and Follow sets and uses them to construct a predictive parsing table for the grammar. The table is to be used to drive a predictive top-down parser. If the input grammar is not LL (1), an appropriate error message should be produced. The generated parser is required to produce some representation of the leftmost derivation for a correct input. If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing.

Phase 3

This phase aims to practice techniques of constructing semantics rules to generate intermediate code. It generates bytecode which follows the standard bytecode instructions defined in Java Virtual Machine Specification. It uses Bison which takes the semantics rule as input. It also uses Flex which takes the lexical file as input and together with Bison output the java bytecode is generated.

Algorithms and Techniques

Phase 1

It uses various classes to do the lexical Analysis

Rules

The Rules class is used to read the rules set and parse it to extract all the rules and separate them into:

1. **Regular Definitions**
2. **Regular Expressions**
3. **Keywords**
4. **Symbols or Punctuations**

The class has some methods, some of them are used to do the logic and some of them are used as getters for the other classes to get the results of the rules logic.

The attributes and data structures used:

1. **vector<pair<string, vector<string>>> rulesData**
    a. It's used to hold the rules data, it's a vector of pairs, each pair consists of:
        i. the type of rule (i.e. keyword, punctuation, regular definition or regular expression).
        ii. A vector of tokens representing the value of the rule.

2. **vector<vector<string>> definitions**
    a. It's used to hold the regular definitions data, it's a vector of vectors, each of which consists of:
        i. The regular definition name as the first element in the vector.
        ii. The value as the second element in the vector.

The classes used:

1. **void readFile(char*)**
    a. Given a file name for the rules set file, it reads the file line by line and sends each line to be parsed using the "parseLine(string)" function.

    b. After parsing every line of the rules we then substitute for every regular definition that is used inside another regular definition using the "substituteDef()" function.

    c. Then we need to substitute for every regular definition used in a regular expression with its value, we do so using the "substituteRules()" function.

2. **void parseLine(string)**
    a. Given a line that is a rule, it parses it into the type of the rule (i.e. keyword, punctuation, regular definition or regular expression) and the actual value of it.

    b. It then adds the extracted information to the "rulesData" attribute if it's not a regular definition and to "definitions" attribute if it's a regular definition.

3. **vector<pair<string, string>> substituteDef()**
    a. It substitutes for every regular definition that is used inside another regular definition.

4. **void substituteRules(vector<pair<string, string>>)**
    a. Given the definitions It substitutes for every regular definition used in a regular expression with its value.

5. **vector<pair<string, vector<string>>> getRulesDate()**
    a. It's used to get the parsed rules data as it returns the "rulesData" attribute.

6. **vector<vector<string>> getDefinitions()**
    a. It's used to get the parsed regular definitions data as it returns the "definitions" attribute.

NFA

Using a struct (trans) to represent the transitions as it contains indexes of states (to & from) and string for the transition weight.

Nfa has a vector for its states and a vector contains the transitions

For constructing an NFA we need for three functions:

1. **Concat**

    used to concatenate two nfas in a single result nfa, as it takes as an input two nfas and return their concatenated nfa

2. **Or_selection**

    This function was used for two purposes, first it is used to generate the or of a given vector of nfas and return the result, and also used to combine multiple nfas in one nfa, as it takes as an input the vector containing the nfas to generate their or nfa or to combine them using a Boolean parameter to specify its purpose

3. **Kleene**

    Used to produce the Kleene of a given nfa and return with the result nfa

    Note that we didn't needed to make a separate function for the + closure as it is the result of concatenated the nfa with its kleene

    The previous functions were the main required functions for constructing a nfa from its regular expression and to combine the resultant nfas in one single nfa

    The next functions are used to adjust the regular expression to match the requirements of the previous functions

4. adjustString

   this function is a helping function as it takes the expression string and adjust it to separate the operations to specify which elements is in an or closure of an and closure

5. interval_substitute

   this function is used to substitute each interval with its logical expression as for example the interval 0-2 is substituted as 0 or 1 or 2 and so on

6. adjust_re

   used to adjust the regular expression given to substitute its intervals or adjust spaces as a helping function

the next functions are used to get the final results

7. re_to_nfa

   this function takes the regular expression as an input and return the generated nfa, note that this function is called for each regular expression to generate its nfa separately using stacks to keep track of current operand and operator

8. input_to_NFAs

   taking a vector of pairs of a string and a vector of strings generated from Rules class

   in which the first string of the pair specifies the type of the regular expression and the vector contains firstly the token name then the regular expressions of the current rule

   this function generates a single nfa for each given rule using the adjusting functions and re_to_nfa function and then combine them in a single result nfa

9. **get_results**

   taking the resultant nfa of the input rules this method generates the final state table of the result nfa to be sent to the DFA class

   the result is produced as a vector of vectors of strings such as a 2d array in which the row number is used for the (from) state and the column number is used for the ascii value of the transition symbol and 0 incase of epsilon, the value of that element is the (to) state\s

10. **get_tokens**

    this function returns a 2d vector of strings to specify the final state and the token of each nfa to be sent to the DFA class, so both transitions table and tokens table are sent to the DFA class

note that the vector of the tokens is filled in function input_to_nfa during the generation of each nfa

DataStructures:

1. **Struct**

   A struct of two integers and a string used to specify the current transition in which two integers for the states indexes (to and from) and a string to store the symbol of the transition

2. **Vector**

   used for transitions table as a 2d vector of strings and a vector of pairs of strings to store final state and token for each nfa, and a vector of pairs of strings and vector of strings to store input rules generated from Rules class

3. **Stack**

   To keep track of operators and operands on converting regular expression to its relative nfa

4. **Pair**

   To store tokens of each final state, and to store current rule in which first element is the type and second element is a vector containing rule name and its regular expressions

DFA

Use 2d vector to represent the transitions between the states the columns are the ASCII of the symbols and the rows are the states that transferred to,

i.e. if NFA_Transition Table[4][48] = 9 it means that the STATE number 4 at the symbol 0 transferred to STATE number 9

Methods:

1. **readParamaters( )**

used to get the parameters from NFA stage that are start State, final States with Tokens of each & 2d vector transition table.

2. **epsilonClosure( )**

used to get the epsilon Closure of all states using stack to get the epsilonClosure of each State and store there in vector of pairs, each state and it's epsilonClosure.

3. **transition( )**

used to get the transitions of the new DFA states through all symbols then get the epsilonClosure of the transition to be union with the transition and finally store it in 2d vector of string called DFA_Transition Table, as there's new states didn't added to the table yet, the algorithm continue till there's no states haven't defined yet and this checked by vector of string that store the finished states.

4. **unionClosure( )**

take a state as a parameter and it is used to get the epsilon Closure it uses the vector of pair and its union the state with its epsilonClosure then removes the duplicates.

5. **convert( )**

used to get the transition of the new added states till there's no states haven't been added yet.

DataStructures:

5. **Vector 2d**

used to store the transition Table.

6. **Stack**

used to get the epsilon Closure of each state by pushing the state that transferred under the EPSILON.

7. **Vector**

used to store the new States in the DFA_Transition Table.

8. **Pair**

Used to store each State with its EpsilonClosure.

DFA MINIMIZATION

It gets the DFA_Transition Table form the previous Stage and it minimize its States

Methods:

1. **minimize ( )**

used to get the parameters from DFA stage that are start State, final States with Tokens of each & DFA_Transition Table & the New States, then it iterate on the states and divide them into two classes according to the states that contains the final State and push the two Classes in vector of String called *TempClasses*.

2. **minimizeTheStates( )**

it uses two vector of String *TempClasses* that's filled previously & *NewClasses* that's empty so far, both vectors in While Loop that only break from it if *TempClasses = NewClasses,* so inside the While Loop there's double iteration through *TempClasses* to get each state with all the others that belong to the same Class through checkBelongToTheSameClass() Function, that push the new classes in the vector *NewClasses* and if *TempClasses = NewClasses* iteration ends.

3. **checkBelongToTheSameClass( )**

used to check if two states belong to the same Class or not, it iterate on the all symbols and iterate on *TempClasses* vector and if the two states transfer to states that belong to same class on *TempClasses* under all symbols then it return 1 and by this both belongs to the same class otherwise it return 0.

4. **minimizedTable ( )**

Firstly, it iterates on all not minimized DFA states and makes a vector of pair to store each state with it's new Minimized State, then it iterates on the new Minimized States and set their transitions in 2d vector DFA_Minimized Table.

5. **finalsWithTokens( )**

used to get all the new final States in the minimized DFA Table and their tokens and store them in a vector of pair.

DataStructures:

1. **2d Vector**

used to store the transition Table.

2. **Vector**

used to store the new States in the DFA_Transition Table.

3. **Pair**

Used to store each State with its new minimized State that used to fill the DFA_Minimized Table.

SIMULATION

The simulation class is used to simulate the input file where the file is read character by character and using the minimized transition table from the DFA Minimization class it transitions from a state to another depending on this character.

The attributes used:

1. **struct token {**
   string token_name;
   string token_value;
   }
   a. The token structure is used consisting of token name and token value

2. **map<string,string> symTable**
   a. The symbol table as a map of string to string, the key is lexeme and the value is the id or keyword initialized with keywords

3. **vector<token> lexemes**
   a. The lexemes as a vector of tokens

4. **Stack**
   a. used to keep track of the transitions of the lexeme

The classes used:

1. **token nextLexeme()**
   a. It reads the next character from the file and moves to the next state using the "move" function.
   b. It then checks if the state reached is a final state using the table of final states and their corresponding token, if so it gets the token corresponding to the reached final state.
   c. Using the stack that keeps track of the lexeme, the last-in final state is the final state of the whole lexeme and if there isn't such one, the lexeme doesn't match any of the patterns

2. **void allLexemes()**
    a. It is the entering method that calls the “nextLexeme” function until the end of the file is reached.

3. **string move(string state, char c)**
    a. Given the current state and a character, it uses the minimized transition table to transition into the next state.

4. **string actionID(string lexeme)**
    a. determines if it keyword or id and populates the symbol table

Output for the given example

int	keyword
sum	id
,	,
count	id
,	,
pass	id
,	,
mnt	id
;	;
while	keyword
(	(
pass	id
!=	relop
10	num
)	)
{	{
pass	id
=	assign
pass	id
+	addop
1	num
;	;
}	}

Phase 2

LL1Grammar

Class description:

This class is responsible for reading and parsing the productions from the input file and convert the given grammar to LL(1) grammar if was not by eliminating the left recursion and left factoring it

It also outputs the list of LHS non-terminals and the productions of the LL(1) grammar to the First and Follow classes

Note that in case of left recursion the new non-terminal production added name is the original production name + "1"

And in case of left factoring the new non-terminal production added name is the original production name + factor name

Functions:

1) read_from_file()

reads the productions from the input file and stores it in vector of strings in which each line contains a production, so if the start of the line wasn't a new non-terminal then it adds it to the previous line as for example:

A = 'a'

| 'b'

Then they are concatenated to # A = 'a' | 'b' to make sure that each element in the vector is a production

2) disassemble_productions()

used to disassemble and extract the production for each line as it separates the left-hand section in a string and the right-hand section elements in a vector of strings after extracting them from the or expression

so, the output is stored as a vector (for multiple productions) of pairs, the first term is a string which is the LHS and the second term is a vector of string which is the RHS terms

3) get_terms(string rhs) && get_non_terminal(string line)

used inside the previous function, as get non terminal function extracts the LHS non-terminal from the current production, and get_terms function splits the RHS around the or sign to get the production RHS terms and store them in a vector of strings

4) substitute_eliminate_lr()

this function is used to substitute in each production by the previous productions and then eliminate the left recursion

the substitution is for eliminating the non-immediate left recursion, so after substituting this function calls the next function to eliminate the left recursion from the current production

5) eliminate_left_recursion(int i)

this function is responsible for eliminating the immediate left recursion for the production with the given index as a parameter

6) left_factoring()

used to left factoring the productions

7) fill_output()

this function fills a vector of strings with the LHS non-terminals and a multimap with each production as the first is the LHS and the second is the term of the RHS

the vector and the multimap are to be sent to the First and Follow class using the getter functions

8) generate_LL1_grammar()

used to call the main functions that generate the LL(1) grammar and fill the required variables to be used later

In addition to some helping functions:

a) **split**: return a vector of strings for the given string split around a given character
b) **erase_SubStr**: used to erase a substring from the string
c) print functions for printing outputs for testing and debugging

Data structures:

- **vector of strings**: to hold input parsed from the input file
- **vector of strings**: to hold the LHS of the productions
- **vector of pairs**: the first element in the pair is the LHS of the current production and the second element is a vector of strings to hold the RHS terms
- **multimap of strings**: to save productions to be sent to First and Follow classes

First

Class description:

This class is responsible for calculating the first of needed terms in the grammar to be used next in the Follow and in constructing the parse table

It takes the LHS vector and the productions multimap from the LL1Grammar class and generates a map for the first of terms in the grammar, in which the first element in the map is the term and the second element is the set of strings containing the first of this term

Functions:

1) get_all_first()

this function gets the output of this class in which it returns a map, its first element is the term and the second element is the set containing its first set

2) get_first(string input)

returns set of first of the input string, it checks for the input to check if first part is a terminal or if it is an epsilon else then it is an expression and it find its first

3) get_single_first(string input)

this function is used by the previous function to get the first of a single term where it can be a terminal or an epsilon or a single non-terminal only

4) first_is_terminal(string s)

used by the previous two functions to check if the given string starts with a terminal so it returns it or an empty string otherwise

5) adjust_reserved_symbols(string& str)

it is a helping function used by first_is_terminal function to adjust the reserved symbols by removing the preceding backslash

6) single_term(string s)

checks if the given string consist of single term only and returns a Boolean value, used by get_first function

7) get_components(string rhs)

this function returns a vector of strings containing the rhs elements of the production after splitting it by the space to get each term

in addition to some helping functions:

a) **split**: return a vector of strings for the given string split around a given character
b) **setters**: to set the input for this class which are the LHS vector and the productions multimap
c) **printers**: to print outputs for testing and debugging purpose

Data structures:

- **multimap of strings**: to save the productions sent from the LL1Grammar class which hold the LHS in the key and the RHS current term in the value
- **vector of strings**: to save the LHS non-terminals
- **set of strings**: to hold current first set
- **map**: to save first set of all elements in which the key is the term string and the value is a set of strings containing the first of this term

Follow

Class description:

This class is responsible for calculating the follow sets of the non-terminals in the grammar to be used next in constructing the parse table.

It takes the LHS vector and the productions multimap from the LL1Grammar class and generates a map for the follow of non-terminals in the grammar, in which the first element in the map is the non-terminal and the second element is the set of strings containing the follow of this non-terminal.

Functions:

1) run()

It is the main entry of the logic done by this class.

It iterates over the LHS vector and uses the iterator value to access the multimap.

It then loops over the returned value of the multimap which each of which is a string of non-terminals and terminals concatenated together.

This string is then splitted into tokens and each token is either a non-terminal or a terminal. A vector of pairs of boolean and string is used to store these tokens which is then sent to computeFollow() function.

2) computeFollow()

This function is responsible for applying the follow rules to the tokens sent to it by the run() function.

When a rule applies it updates the map of sets representing the non-terminal and its follow set with the new values obtained by applying the rule.

It uses the first sets obtained from the First class in order to update the follow set of the non-terminals.

3) getFollowSets()

This function is responsible for returning the computed follow sets of the non-terminals to be used in constructing the parsing table.

Data structures:

- **multimap of strings**: to save the productions sent from the LL1Grammar class which hold the LHS in the key and the RHS current term in the value
- **vector of strings**: to save the LHS non-terminals
- **vector of pairs of boolean and string**: to save the tokens obtained by splitting the each entry of the multimap of a LHS in which the boolean indicates the type of the token, false for non-terminals and true for terminals, and the string is the token itself.
- **map of sets**: to save the first set of all elements obtained from the First class in which the key is the term string and the value is a set of strings containing the first of this term.
- **map of sets**: to save the follow set of all non-terminals in which the key is the term string and the value is a set of strings containing the follow set of this non-terminal.

ParsingTable

Class description:

ParsingTable is responsible for constructing the Predictive Parsing Table, and parsing the input string with the help of the Lexical Analyzer. It takes the productions, the starting symbol, the first sets, the follow sets, and the lexical analyzer and generates the predictive parsing table, and the leftmost derivation of the input.

Functions:

1) **Constructor(lexicalAnalyzer, &production, &first, &follow)**

runs the table construction process, and parse the string specified by the lexical analyzer.

2) **buildTable()**

constructs the predictive parsing table. The algorithm follows directly from [Dragon's Book], Algorithm 4.31. The table is printed in "table.txt".

3) **addSynch()**

adds the "synch" entries in the table. We choose the follow sets to be the synchronizing sets.

4) **printTable()**

prints the predictive parsing table in the form: non-terminal.terminal --> entry

5) **queryTable(string nonTerminal, string terminal)**

returns the entry specified by the nonTerminal and terminal.

6) **parse()**

parses the string input specified by the lexical analyzer. The derivation is printed in "derivation.txt"

Decisions: '$' represents the end of a string.

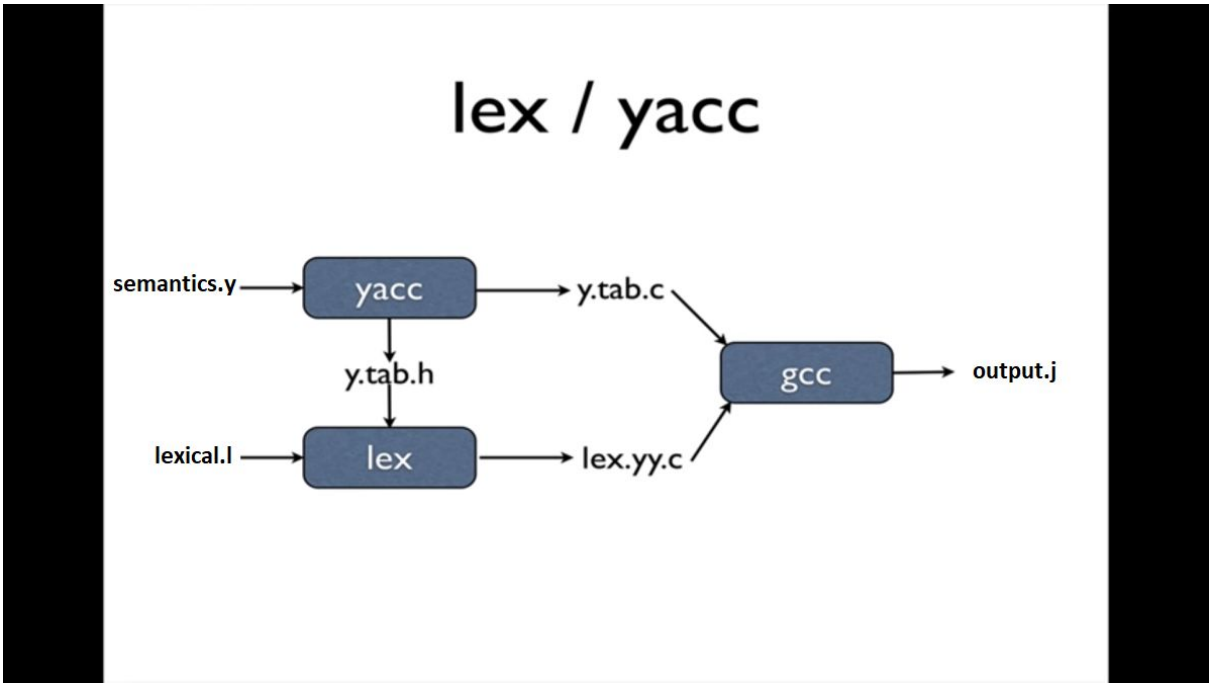
deque derivationStack: We choose a deque, so that we can directly access the first element -just like a normal stack-, in addition to be able to print the content of the list.

Algorithm follows directly from [Dragon's Book], Algorithm 4.34. The next token is provided by lexicalAnalyzer::nextLexeme().

Data structures:

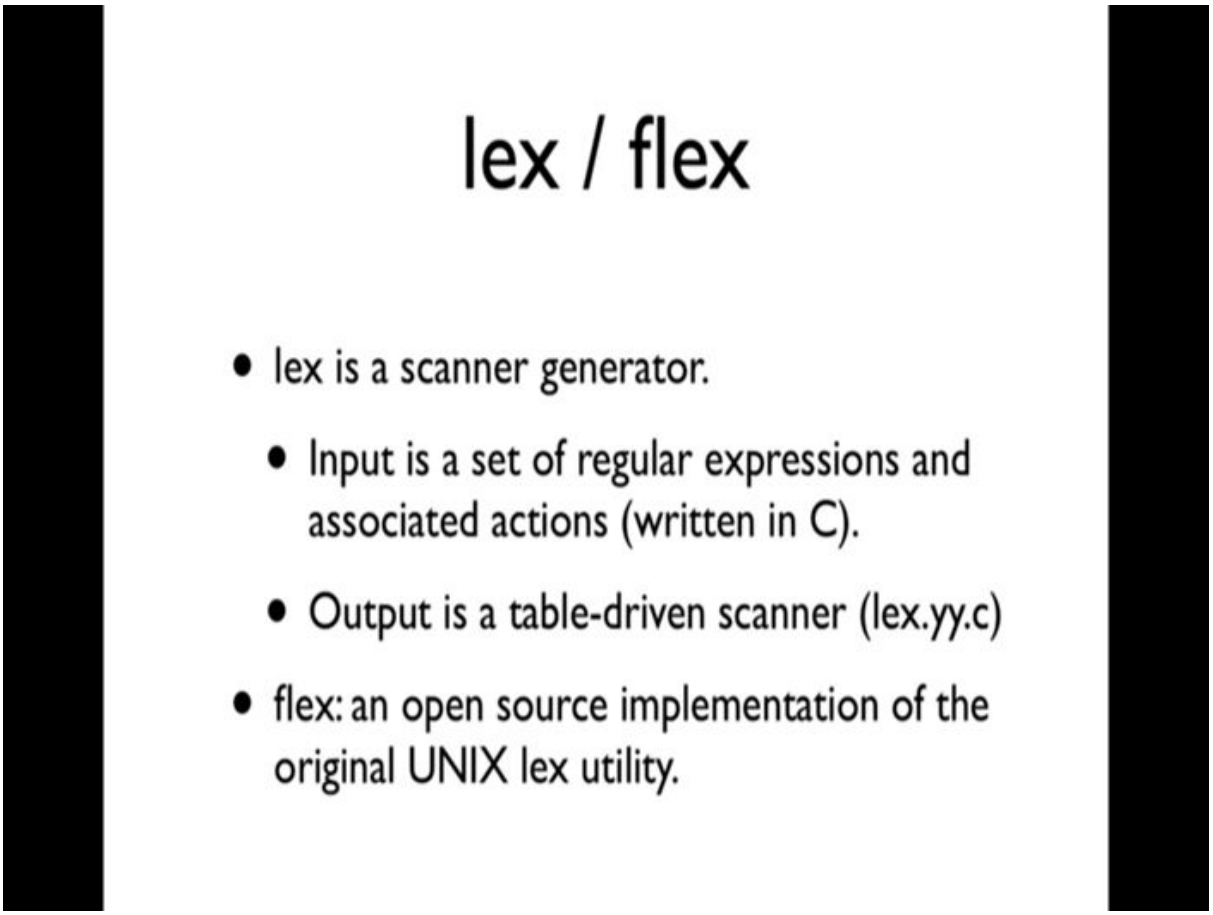
- **map<string,string>**: represents the parsing table. We choose a map to provide fast access (lg N), and no space is wasted as in the case of using an array.

Phase 3



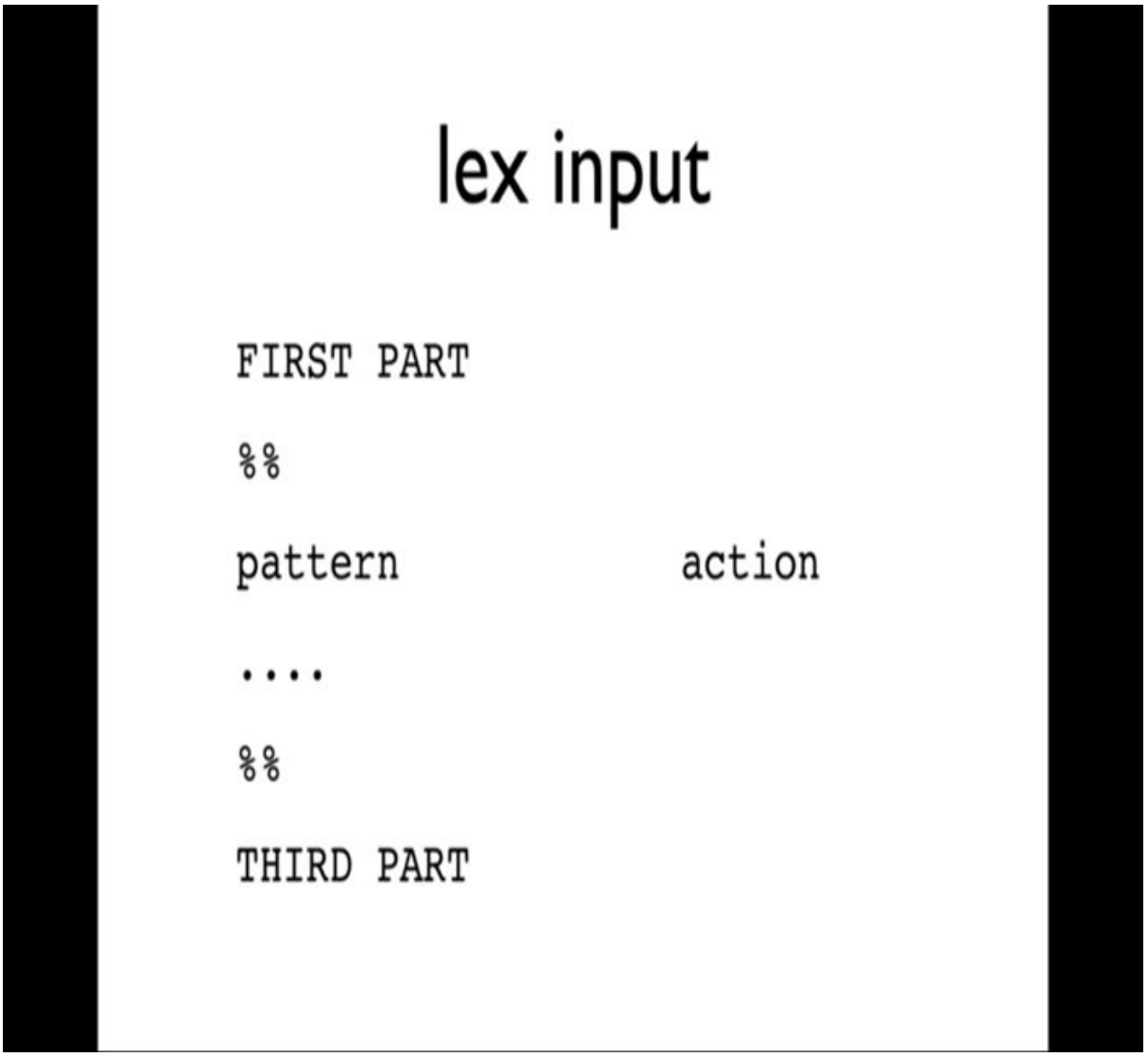
Lexical.l:

description:



The class contains the tokens of the language and the associated action with each given token

format:



Tokens declaration:

```
11    letter [a-zA-Z]
12    digit [0-9]
13    id {letter}({letter}|{digit})*
14    digits {digit}+
15    num "-"?{digits}
16    fNum "-"?{digits}.{digits}("E"{digits})?
17    relop "=="|"!="|">"|">="|"<"|"<="
18    boolop "&&"|"||"
19    // add all ops together
20    op "+"|"-"|"*"|"/"|"%"|"&"|"|"
21    boolVal "true"|"false"
22
```

Regular Expressions with Actions:

```
23  %%
24
25  /*              keywords          */
26  "int"   {return INT;}
27  "float" {return FLOAT;}
28  "boolean"   {return BOOLEAN;}
29  "if"    {return IF;}
30  "else"  {return ELSE;}
31  "while" {return WHILE;}
32  "for"   {return FOR;}
33
34  /*                 special characters                */
35  ";" {return SEMI_COLON;}
36  "," {return COMMA;}
37  "(" {return LEFT_BRACKET;}
38  ")" {return RIGHT_BRACKET;}
39  "{" {return LEFT_CURLY_BRACES;}
40  "}" {return RIGHT_CURLY_BRACES;}
41  "=" {return EQUAL;}
42  "System.out.println"    {return SYSTEM_OUT;}
43
44  // increment lineCounter
45  \n {lineCounter++; printLineNo(lineCounter);}
46
47  /*                 actions               */
48  {op}    {yylval.aopval = strdup(yytext); return ARITH_OP;}
49  {num}   {yylval.ival = atoi(yytext); return INT;}
50  {fNum}  {yylval.fval = atof(yytext); return FLOAT;}
51  {relop} {yylval.aopval = strdup(yytext); return RELA_OP;}
52  {boolop} {yylval.aopval = strdup(yytext); return BOOL_OP;}
53  {boolVal} {if(!strcmp(yytext,"true")){ yylval.bval = 1;} else { yylval.bval = 0;} return BOOL;}
54  {id}    {yylval.idval = strdup(yytext);return IDENTIFIER;}
55
56  %%
```

Lexical.l contains:

- Function called yylex( ) that returns 1 whenever an expression you have specified is found in the input text, 0 when end of file is encountered, each call to yylex( ) parses one token and when it called again, it picks up where is left off.

- Variable called yyval that is defined as C union having a member called yytext to point to character strings, also :

  1. ival to hold an integer value using atoi( ) function in C
  2. fval to hold a float value using atof( ) function in C
  3. aopval to hold arithmetic operation using strdup( ) function in C

  These members were performed in Yacc specification in Union.

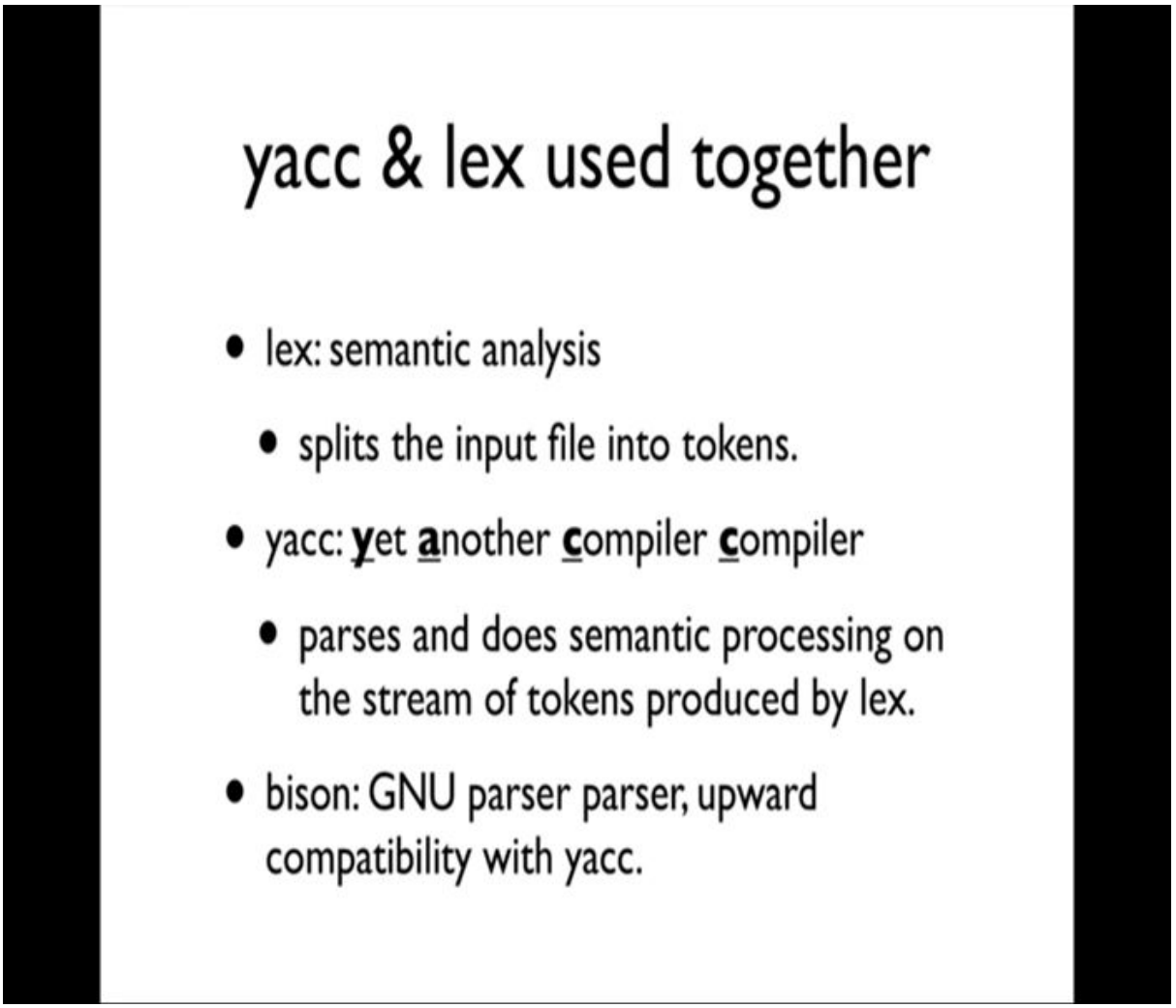
- Function yywrap( ) that is implemented in the class, that's used in case of multiple input files that is called by Lex.

```
58  int yywrap() {
59      // handle IOF
60      return -1;
61  }
```

Semantics.y:

description:

The class contains the productions and it is semantic Rules for Parsing



Format:

yacc - first part

- first part of a yacc specification includes:
  - C declarations enclosed in %{ %}
  - yacc definitions
    - %start
    - %token
    - %union
    - %type

yacc - productions

- The middle section represents a grammar - a set of productions. The left-hand side of a production is followed by a colon, and a right hand side.
- Multiple right-hand sides may follow separated by a '|'
- Actions associated with a rule are entered in braces.

yacc productions

- $1, $2….$n can be refer to the values associated with symbols
- $$ refer to the value of the left
- Every symbol have a value associated with it (including token and non-terminals)
- Default action:
  - $$ = $1

yacc - third part

- contains valid C code that supports the language processing.
  - symbol table implementation
  - functions that might be called by actions associated with the productions in the second part

Data Structure:

- Map<String, pair<int, types>> it is the structure of Symbol Table.
- Enum {} called types that store all primitive data types (INT, FLOAT, ...).
- Map<String, String>, used to map each operation to its name EX: {+, 'add'}.
- Vector<String>, called bytecodeList contains the generated JAVA ByteCode.
- int LabelsCount, used to count the labels in the program.

Functions:

- generateHeader()

  build the header of JAVA Byte Code file by adding specific lines to the bytecodeList vector.

- generateFooter()

  build the footer of JAVA Byte Code file by adding specific lines to the bytecodeList vector like return statement.

- checkCast(int x, int y, string op)

  check if x = y, then check if x is INTEGER type, add in the bytecodeList (i) & the Name of the operation op but if x is FLOAT type, add in the bytecodeList (f) & the Name of the operation op, otherwise it throws error.

- backPatch(vector<int> List, int num)

  it fills up unspecified information of labels when it reaches to the jumped lines using a vector that contains the next lines that will jump to and number of Label by iterating on bytecodeList.

- checkOp(String id)

  return bool, it checks whether the symbol table contains the string or not.

- defineVar(String name, int type)

  check if the string in Symbol Table if it exists, throw yyerror() else, if type is INTEGER add in bytecodeList (iconst_0) and name and if type is FLOAT add in bytecodeList (fconst_0)
  In the Both cases add the String in the Symbol Table with its type.

- generateLabel()

  return L_labelsCount that added to the bytecodeList.

- merge(vector<int> List1, vector<int> List2)

  merge the two lists and return the merged list

Productions:

```
label:
    {   $$ = labelsCount;   writeCode(generateLabel() + ":");    }
    ;
```

Generate L_(num) in Java Byte Code to indicate each line in the program that can be used in jumps instead of addresses and PC counter using labelsCount that's a global variable that is incremented at each call.

```
goto:
    {   $$ = bytecodeList.size();   writeCode("goto "); }
    ;
```

Generate goto: in Java Byte Code to indicate that jump happens and after that the jumped Label is written after it.

```
primitive_type:
    INT_SYM {$$ = INTEGER_TYPE;}
    | FLOAT_SYM {$$ = FLOAT_TYPE;}
    |BOOLEAN_SYM {$$ = BOOLEAN_TYPE;}
    ;
```

Its Attribute that is specify type of Identifier whether int, float, boolean.

```
declaration:
    primitive_type IDENTIFIER SEMI_COLON
    {   string str($2);
        if($1 == INTEGER_TYPE)
        {
            defineVar(str,INTEGER_TYPE);
        }else if ($1 == FLOAT_TYPE)
        {
            defineVar(str,FLOAT_TYPE);
        }
    }
    ;
```

“int x;”
Declare variable by adding it in Symbol Table and specifying variable Name with its type.

```
statement:
    declaration {vector<int> * v = new vector<int>(); $$.nextList =v;}
    |if {$$.nextList = $1.nextList;}
    |while  {$$.nextList = $1.nextList;}
    |for {$$.nextList = $1.nextList;}
    | assignment {vector<int> * v = new vector<int>(); $$.nextList =v;}
    | system_print {vector<int> * v = new vector<int>(); $$.nextList =v;}
    ;
```

Assign its next list to the next list of the production in case of (if, while, for) and assign empty next list otherwise.

```
statement_list:
    statement
    |
    statement
    label
    statement_list
    {   backPatch($1.nextList,$2);  $$.nextList = $3.nextList;  }
    ;
```

Perform backPatch on the next list of statement with statement_list and assign the next list to the next list of current statement_list.

```
assignment:
    IDENTIFIER EQUAL expression SEMI_COLON
    {
        string str($1);
        // store top of stack to the id
        if(checkOp(str))
        {
            if($3.sType == symbTab[str].second)
            {
                if($3.sType == INTEGER_TYPE)
                {
                    writeCode("istore " + to_string(symbTab[str].first));
                }else if ($3.sType == FLOAT_TYPE)
                {
                    writeCode("fstore " + to_string(symbTab[str].first));
                }
            }
            else
            {
                yyerror("type mismatch, cast required");
            }
        }else{
            string err = "identifier: "+str+" isn't declared in this scope";
            yyerror(err.c_str());
        }
    }
    ;
```

“x = 5;”
Assign value to Identifier, associated with following action

- check if identifier is predeclared in Symbol Table, otherwise release error of undeclaration.
- check type of identifier & expression, if matches then it writes in Java byte code by storing of value Otherwise release error datatype mismatch.

```
expression:
    FLOAT   {$$.sType = FLOAT_TYPE; writeCode("ldc "+to_string($1));}
    | INT   {$$.sType = INTEGER_TYPE;  writeCode("ldc "+to_string($1));}
    | expression ARITH_OP expression    {checkCast($1.sType, $3.sType, string($2));}
    | IDENTIFIER {
        string str($1);
        if(checkOp(str))
        {
            $$.sType = symbTab[str].second;
            if(symbTab[str].second == INTEGER_TYPE)
            {
                writeCode("iload " + to_string(symbTab[str].first));
            }else if (symbTab[str].second == FLOAT_TYPE)
            {
                writeCode("fload " + to_string(symbTab[str].first));
            }
        }
        else
        {
            string err = "identifier: "+str+" isn't declared in this scope";
            yyerror(err.c_str());
            $$.sType = ERROR_TYPE;
        }
    }
    | LEFT_BRACKET expression RIGHT_BRACKET {$$.sType = $2.sType;}
    ;
```

Assign type of expression to

- int or float.
- in case of arithmetic operation between two expressions, check cast
- in case of identifier, check whether exist in symbol table or not and assign its type to the identifier type otherwise release error.

```
b_expression:
    BOOLEAN
    {
        if($1)  // "true"
        {
            $$.trueList = new vector<int> ();
            $$.trueList->push_back(bytecodeList.size());
            $$.falseList = new vector<int>();
            writeCode("goto ");
        }else
        {
            $$.trueList = new vector<int> ();
            $$.falseList= new vector<int>();
            $$.falseList->push_back(bytecodeList.size());
            writeCode("goto ");
        }
    }
    |b_expression BOOL_OP label b_expression
    {
        if(!strcmp($2, "&&"))
        {
            backPatch($1.trueList, $3);
            $$.trueList = $4.trueList;
            $$.falseList = merge($1.falseList,$4.falseList);
        }
        else if (!strcmp($2,"||"))
        {
            backPatch($1.falseList,$3);
            $$.trueList = merge($1.trueList, $4.trueList);
            $$.falseList = $4.falseList;
        }
    }
    | expression RELA_OP expression
```

b_expression can be just a Boolean value (true or false) associated with a goto and add its index in code list to trueList to specify its label
it can be two Boolean expressions and perform a boolean operation on them, whether it is AND or OR operations, and in both cases back patching is used to specify which label to jump to according to conditions results

```
| expression RELA_OP expression
{
    string op ($2);
    $$.trueList = new vector<int>();
    $$.trueList ->push_back (bytecodeList.size());
    $$.falseList = new vector<int>();
    $$.falseList->push_back(bytecodeList.size()+1);
    writeCode(getOp(op)+ " ");
    writeCode("goto ");
}
;
```

Finally, boolean expressions can be two expressions and performing relation operations on them, associated with a goto and add its index in code list to trueList and falseList to specify its label.

```
if:
    IF_SYM LEFT_BRACKET
    b_expression
    RIGHT_BRACKET LEFT_CURLY_BRACES
    label
    statement_list
    goto
    RIGHT_CURLY_BRACES
    ELSE_SYM LEFT_CURLY_BRACES
    label
    statement_list
    RIGHT_CURLY_BRACES
    {
        backPatch($3.trueList,$6);
        backPatch($3.falseList,$12);
        $$.nextList = merge($7.nextList, $13.nextList);
        $$.nextList->push_back($8);
    }
    ;
```

Perform backPatch on

- True list of the boolean expressions with body of true condition.
- False list of the boolean expression with body of false condition.
- Assign next list the merging of both cases as it doesn't know which Label will jump to.

```
while:
    label
    WHILE_SYM LEFT_BRACKET
    b_expression
    RIGHT_BRACKET LEFT_CURLY_BRACES
    label
    statement_list
    RIGHT_CURLY_BRACES
    {
        writeCode("goto " + getLabel($1));
        backPatch($8.nextList,$1);
        backPatch($4.trueList,$7);
        $$.nextList = $4.falseList;
    }
    ;
```

Perform backPatch on

- Next list of while loop body with condition of while loop to repeat the loop again.

- True list of while condition with while body to decide whether enter while loop or not adding the false List of boolean expressions in while next List to terminate the while Loop.

```
for:
    FOR_SYM LEFT_BRACKET assignment
    label b_expression SEMI_COLON
    label assignment goto RIGHT_BRACKET
    LEFT_CURLY_BRACES label statement_list goto RIGHT_CURLY_BRACES
    {
        backPatch($5.trueList,$12);
        vector<int> * v = new vector<int> ();
        v->push_back($9);
        backPatch(v,$4);
        v = new vector<int>();
        v->push_back($14);
        backPatch(v,$7);
        backPatch($13.nextList,$7);
        $$.nextList = $5.falseList;
    }
    ;
```

Perform backPatch on

- True list of for loop conditions with for loop body to decide whether enter for loop or not.
- Assign new value of counter with assignment of counter and check validity of boolean expression.
- End of For loop body with counter, increment counter.
- Next list of Body of For loop with counter adding the false List of boolean expression in for loop next List to terminate the for Loop.

```
system_print:
SYSTEM_OUT LEFT_BRACKET expression RIGHT_BRACKET SEMI_COLON
{
    if($3.sType == INTEGER_TYPE)
    {
        // standard java bytecode for system out statement
        // expression is at top of stack now
        // save it at the predefined temp syso var
        writeCode("istore " + to_string(symbTab["1syso_int_var"].first));
        writeCode("getstatic      java/lang/System/out Ljava/io/PrintStream;");
        writeCode("iload " + to_string(symbTab["1syso_int_var"].first ));
        writeCode("invokevirtual java/io/PrintStream/println(I)V");

    }else if ($3.sType == FLOAT_TYPE)
    {
        writeCode("fstore " + to_string(symbTab["1syso_float_var"].first));
        writeCode("getstatic      java/lang/System/out Ljava/io/PrintStream;");
        writeCode("fload " + to_string(symbTab["1syso_float_var"].first ));
        writeCode("invokevirtual java/io/PrintStream/println(F)V");
    }
}
;
```

To print expression using System.out.ln(expression);
Standard system out java bytecode is written depending on the value of the expression.

Results

Phase 1

For the given input:

```
int sum , count , pass , mnt; while (pass != 10)
{
pass = pass + 1 ;
}
```

Output:

```
int             keyword
sum             id
,               ,
count           id
,               ,
pass            id
,               ,
mnt             id
;               ;
while           keyword
(               (
pass            id
!=              relop
10              num
)               )
{               {
pass            id
=               assign
pass            id
+               addop
1               num
;               ;
}               }
```

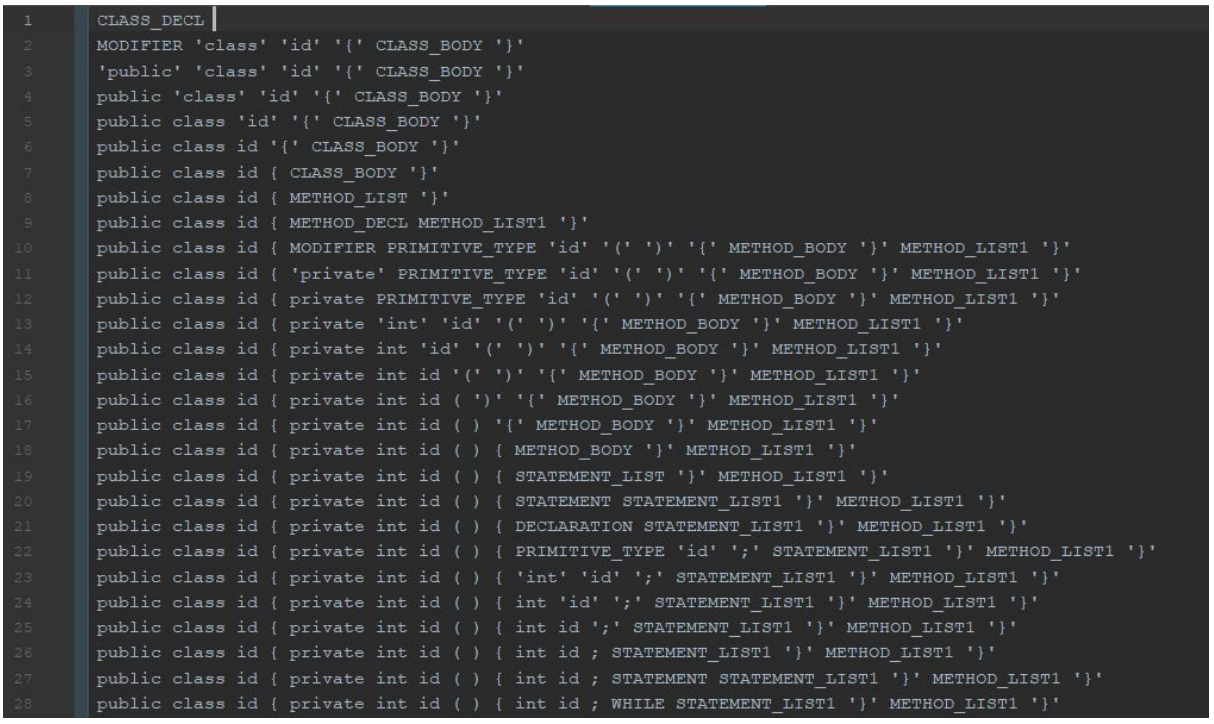
Phase 2

For the given input and productions:

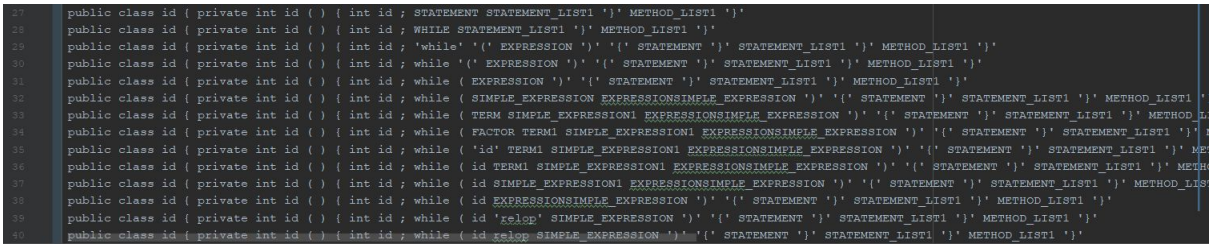
```
letter = a-z | A-Z
digit = 0 - 9
id: letter (letter|digit)*
digits = digit+
{boolean int float}
num: digit+ | digit+ . digit+ ( \L | E digit+)
relop: \=\= | !\= | > | >\= | < | <\=
assign: \=
{if else while int float private public class}
[; , \( \) \{ \}]
addop: \+ | -
mulop: \* | /
```

```
# CLASS_DECL = MODIFIER 'class' 'id' '{' CLASS_BODY '}'
# CLASS_BODY = DECLARATION | ASSIGNMENT | METHOD_LIST | \L
# METHOD_LIST = METHOD_DECL | METHOD_LIST METHOD_DECL
# METHOD_DECL = MODIFIER PRIMITIVE_TYPE 'id' '(' ')' '{' METHOD_BODY '}'
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT = DECLARATION | IF | WHILE | ASSIGNMENT
# DECLARATION = PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE = 'int' | 'float'
# MODIFIER = 'private' | 'public'
# 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 = '+' | '-'
```

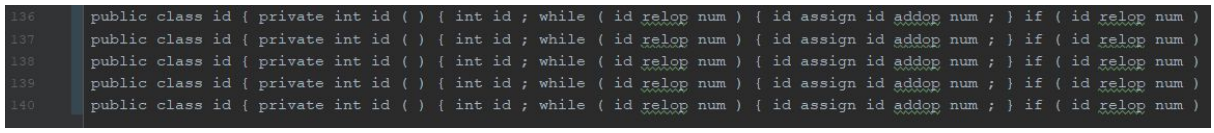
Snapshots of leftmost derivation:



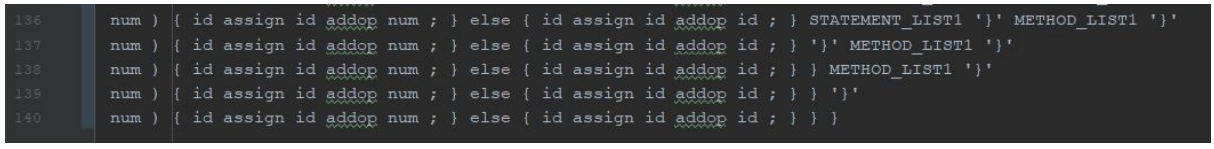
```
CLASS_DECL
MODIFIER 'class' 'id' '{' CLASS_BODY '}'
'public' 'class' 'id' '{' CLASS_BODY '}'
public 'class' 'id' '{' CLASS_BODY '}'
public class 'id' '{' CLASS_BODY '}'
public class id '{' CLASS_BODY '}'
public class id { CLASS_BODY '}'
public class id { METHOD_LIST '}'
public class id { METHOD_DECL METHOD_LIST1 '}'
public class id { MODIFIER PRIMITIVE_TYPE 'id' '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { 'private' PRIMITIVE_TYPE 'id' '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private PRIMITIVE_TYPE 'id' '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private 'int' 'id' '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int 'id' '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int id '(' ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int id ( ')' '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int id ( ) '{' METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int id ( ) { METHOD_BODY '}' METHOD_LIST1 '}'
public class id { private int id ( ) { STATEMENT_LIST '}' METHOD_LIST1 '}'
public class id { private int id ( ) { STATEMENT STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { DECLARATION STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { PRIMITIVE_TYPE 'id' ';' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { 'int' 'id' ';' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int 'id' ';' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ';' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; STATEMENT STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; WHILE STATEMENT_LIST1 '}' METHOD_LIST1 '}'
```



```
public class id { private int id ( ) { int id ; STATEMENT STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; WHILE STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; 'while' '(' EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; while '(' EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; while ( EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; while ( SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '
public class id { private int id ( ) { int id ; while ( TERM SIMPLE_EXPRESSION1 EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_L
public class id { private int id ( ) { int id ; while ( FACTOR TERM1 SIMPLE_EXPRESSION1 EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}'
public class id { private int id ( ) { int id ; while ( 'id' TERM1 SIMPLE_EXPRESSION1 EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' ME
public class id { private int id ( ) { int id ; while ( id TERM1 SIMPLE_EXPRESSION1 EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METH
public class id { private int id ( ) { int id ; while ( id SIMPLE_EXPRESSION1 EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIS
public class id { private int id ( ) { int id ; while ( id EXPRESSIONSIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; while ( id 'relop' SIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
public class id { private int id ( ) { int id ; while ( id relop SIMPLE_EXPRESSION ')' '{' STATEMENT '}' STATEMENT_LIST1 '}' METHOD_LIST1 '}'
```



```
public class id { private int id ( ) { int id ; while ( id relop num ) { id assign id addop num ; } if ( id relop num )
public class id { private int id ( ) { int id ; while ( id relop num ) { id assign id addop num ; } if ( id relop num )
public class id { private int id ( ) { int id ; while ( id relop num ) { id assign id addop num ; } if ( id relop num )
public class id { private int id ( ) { int id ; while ( id relop num ) { id assign id addop num ; } if ( id relop num )
public class id { private int id ( ) { int id ; while ( id relop num ) { id assign id addop num ; } if ( id relop num )
```



```
num ) { id assign id addop num ; } else { id assign id addop id ; } STATEMENT_LIST1 '}' METHOD_LIST1 '}'
num ) { id assign id addop num ; } else { id assign id addop id ; } '}' METHOD_LIST1 '}'
num ) { id assign id addop num ; } else { id assign id addop id ; } } METHOD_LIST1 '}'
num ) { id assign id addop num ; } else { id assign id addop id ; } } '}'
num ) { id assign id addop num ; } else { id assign id addop id ; } } }
```

Phase 3

For input file:

```
int x ;
x = 50;
if ( x == 50 )
{
    x = 200;
}
else
{
    x= 20;
}
System.out.println(x);
int i;
for(i=0; i<10; i=i+1;){
    System.out.println(i);
}
while(i >= 0){
    System.out.println(i);
    i = i - 1;
}
```

Output result:

```
1   .source input.txt
2   .class public test
3   .super java/lang/Object
4
5   .method public <init>()V
6   aload_0
7   invokenonvirtual java/lang/Object/<init>()V
8   return
9   .end method
10
11  .method public static main([Ljava/lang/String;)V
12  .limit locals 100
13  .limit stack 100
14  iconst_0
15  istore 1
```

```
16   fconst_0
17   fstore 2
18   .line 1
19   iconst_0
20   istore 3
21   .line 2
22   L_0:
23   ldc 50
24   istore 3
25   .line 3
26   L_1:
27   iload 3
28   ldc 50
29   if_icmpeq L_2
30   goto L_3
31   .line 4
32   L_2:
33   .line 5
34   ldc 200
35   istore 3
36   .line 6
37   goto L_4
38   .line 7
39   .line 8
40   L_3:
41   .line 9
42   ldc 20
43   istore 3
44   .line 10
45   .line 11
46   L_4:
47   iload 3
48   istore 1
49   getstatic      java/lang/System/out Ljava/io/PrintStream;
50   iload 1
51   invokevirtual java/io/PrintStream/println(I)V
52   .line 12
53   L_5:
54   iconst_0
55   istore 4
```

```
56    .line 13
57    L_6:
58    ldc 0
59    istore 4
60    L_7:
61    iload 4
62    ldc 10
```

```
63   if_icmplt L_9
64   goto L_10
65   L_8:
66   iload 4
67   ldc 1
68   iadd
69   istore 4
70   goto L_7
71   L_9:
72   .line 14
73   iload 4
74   istore 1
75   getstatic      java/lang/System/out Ljava/io/PrintStream;
76   iload 1
77   invokevirtual java/io/PrintStream/println(I)V
78   .line 15
79   goto L_8
80   .line 16
81   L_10:
82   L_11:
83   iload 4
84   ldc 0
85   if_icmpge L_12
86   goto L_14
87   L_12:
88   .line 17
89   iload 4
90   istore 1
91   getstatic      java/lang/System/out Ljava/io/PrintStream;
92   iload 1
93   invokevirtual java/io/PrintStream/println(I)V
94   .line 18
95   L_13:
96   iload 4
97   ldc 1
98   isub
99   istore 4
100  .line 19
101  goto L_11
102  L_14:
```

```
103    return
104    .end method
105
```

Minimized DFA Table

Snapshots of minimized dfa table as follow:

```
from 1  on getting 61 to 55
from 2  on getting 48 to 7
from 2  on getting 49 to 7
from 2  on getting 50 to 8
from 2  on getting 51 to 8
from 2  on getting 52 to 9
from 2  on getting 53 to 9
from 2  on getting 54 to 10
from 2  on getting 55 to 10
from 2  on getting 56 to 11
from 2  on getting 57 to 11
from 3  on getting 48 to 7
from 3  on getting 49 to 7
from 3  on getting 50 to 8
from 3  on getting 51 to 8
from 3  on getting 52 to 9
from 3  on getting 53 to 9
from 3  on getting 54 to 10
from 3  on getting 55 to 10
from 3  on getting 56 to 11
from 3  on getting 57 to 11
from 4  on getting 48 to 7
from 4  on getting 49 to 7
from 4  on getting 50 to 8
from 4  on getting 51 to 8
from 4  on getting 52 to 9
from 4  on getting 53 to 9
from 4  on getting 54 to 10
from 4  on getting 55 to 10
from 4  on getting 56 to 11
from 4  on getting 57 to 11
from 5  on getting 48 to 7
from 5  on getting 49 to 7
from 5  on getting 50 to 8
from 5  on getting 51 to 8
from 5  on getting 52 to 9
from 5  on getting 53 to 9
from 5  on getting 54 to 10
from 5  on getting 55 to 10
from 5  on getting 56 to 11
from 5  on getting 57 to 11
from 6  on getting 48 to 7
from 6  on getting 49 to 7
```

```
from 6  on getting 49 to 7
from 6  on getting 50 to 8
from 6  on getting 51 to 8
from 6  on getting 52 to 9
from 6  on getting 53 to 9
from 6  on getting 54 to 10
from 6  on getting 55 to 10
from 6  on getting 56 to 11
from 6  on getting 57 to 11
from 7  on getting 46 to 90
from 7  on getting 48 to 12
from 7  on getting 49 to 12
from 7  on getting 50 to 13
from 7  on getting 51 to 13
from 7  on getting 52 to 14
from 7  on getting 53 to 14
from 7  on getting 54 to 15
from 7  on getting 55 to 15
from 7  on getting 56 to 16
from 7  on getting 57 to 16
from 8  on getting 46 to 90
from 8  on getting 48 to 12
from 8  on getting 49 to 12
from 8  on getting 50 to 13
from 8  on getting 51 to 13
from 8  on getting 52 to 14
from 8  on getting 53 to 14
from 8  on getting 54 to 15
from 8  on getting 55 to 15
from 8  on getting 56 to 16
from 8  on getting 57 to 16
from 9  on getting 46 to 90
from 9  on getting 48 to 12
from 9  on getting 49 to 12
from 9  on getting 50 to 13
from 9  on getting 51 to 13
from 9  on getting 52 to 14
from 9  on getting 53 to 14
from 9  on getting 54 to 15
from 9  on getting 55 to 15
from 9  on getting 56 to 16
from 9  on getting 57 to 16
from 10  on getting 46 to 90
```

```
from 10  on getting 48 to 12
from 10  on getting 49 to 12
from 10  on getting 50 to 13
from 10  on getting 51 to 13
from 10  on getting 52 to 14
from 10  on getting 53 to 14
from 10  on getting 54 to 15
from 10  on getting 55 to 15
from 10  on getting 56 to 16
from 10  on getting 57 to 16
from 11  on getting 46 to 90
from 11  on getting 48 to 12
from 11  on getting 49 to 12
from 11  on getting 50 to 13
from 11  on getting 51 to 13
from 11  on getting 52 to 14
from 11  on getting 53 to 14
from 11  on getting 54 to 15
from 11  on getting 55 to 15
from 11  on getting 56 to 16
from 11  on getting 57 to 16
from 12  on getting 46 to 90
from 12  on getting 48 to 12
from 12  on getting 49 to 12
from 12  on getting 50 to 13
from 12  on getting 51 to 13
from 12  on getting 52 to 14
from 12  on getting 53 to 14
from 12  on getting 54 to 15
from 12  on getting 55 to 15
from 12  on getting 56 to 16
from 12  on getting 57 to 16
from 13  on getting 46 to 90
from 13  on getting 48 to 12
from 13  on getting 49 to 12
from 13  on getting 50 to 13
from 13  on getting 51 to 13
from 13  on getting 52 to 14
from 13  on getting 53 to 14
from 13  on getting 54 to 15
from 13  on getting 55 to 15
from 13  on getting 56 to 16
from 13  on getting 57 to 16
from 14  on getting 46 to 90
from 14  on getting 48 to 12
from 14  on getting 49 to 12
from 14  on getting 50 to 13
from 14  on getting 51 to 13
from 14  on getting 52 to 14
from 14  on getting 53 to 14
from 14  on getting 54 to 15
from 14  on getting 55 to 15
from 14  on getting 56 to 16
from 14  on getting 57 to 16
from 15  on getting 46 to 90
from 15  on getting 48 to 12
from 15  on getting 49 to 12
from 15  on getting 50 to 13
from 15  on getting 51 to 13
from 15  on getting 52 to 14
from 15  on getting 53 to 14
from 15  on getting 54 to 15
from 15  on getting 55 to 15
from 15  on getting 56 to 16
from 15  on getting 57 to 16
from 16  on getting 46 to 90
from 16  on getting 48 to 12
from 16  on getting 49 to 12
from 16  on getting 50 to 13
from 16  on getting 51 to 13
from 16  on getting 52 to 14
from 16  on getting 53 to 14
from 16  on getting 54 to 15
from 16  on getting 55 to 15
from 16  on getting 56 to 16
from 16  on getting 57 to 16
from 17  on getting 33 to 1
from 17  on getting 40 to 18
from 17  on getting 41 to 19
from 17  on getting 42 to 17
from 17  on getting 43 to 20
from 17  on getting 44 to 21
from 17  on getting 45 to 22
from 17  on getting 47 to 17
```

Parsing Table

Parsing table is in the format: non-terminal.topInInput -> entry

The missing entries represent the "error" entries.

The synchronizing sets are represented by "synch" entries.

```
ASSIGNMENT.$ --> synch
ASSIGNMENT.float --> synch
ASSIGNMENT.id --> 'id' '=' EXPRESSION ';'
ASSIGNMENT.if --> synch
ASSIGNMENT.int --> synch
ASSIGNMENT.while --> synch
ASSIGNMENT.} --> synch
DECLARATION.$ --> synch
DECLARATION.float --> PRIMITIVE_TYPE 'id' ';'
DECLARATION.id --> synch
DECLARATION.if --> synch
DECLARATION.int --> PRIMITIVE_TYPE 'id' ';'
DECLARATION.while --> synch
DECLARATION.} --> synch
EXPRESSION.( --> SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION
EXPRESSION.) --> synch
EXPRESSION.+ --> SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION
EXPRESSION.- --> SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION
EXPRESSION.; --> synch
EXPRESSION.id --> SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION
EXPRESSION.num --> SIMPLE_EXPRESSION EXPRESSIONSIMPLE_EXPRESSION
EXPRESSIONSIMPLE_EXPRESSION.) --> \L
EXPRESSIONSIMPLE_EXPRESSION.; --> \L
EXPRESSIONSIMPLE_EXPRESSION.relop --> 'relop' SIMPLE_EXPRESSION
FACTOR.( --> '(' EXPRESSION ')'
FACTOR.) --> synch
FACTOR.; --> synch
FACTOR.addop --> synch
FACTOR.id --> 'id'
FACTOR.mulop --> synch
FACTOR.num --> 'num'
FACTOR.relop --> synch
IF.$ --> synch
IF.float --> synch
IF.id --> synch
IF.if --> 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'
IF.int --> synch
IF.while --> synch
IF.} --> synch
```

```
METHOD_BODY.$ --> synch
METHOD_BODY.float --> STATEMENT_LIST
METHOD_BODY.id --> STATEMENT_LIST
METHOD_BODY.if --> STATEMENT_LIST
METHOD_BODY.int --> STATEMENT_LIST
METHOD_BODY.while --> STATEMENT_LIST
PRIMITIVE_TYPE.float --> 'float'
PRIMITIVE_TYPE.id --> synch
PRIMITIVE_TYPE.int --> 'int'
SIGN.( --> synch
SIGN.+ --> '+'
SIGN.- --> '-'
SIGN.id --> synch
SIGN.num --> synch
SIMPLE_EXPRESSION.( --> TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION.) --> synch
SIMPLE_EXPRESSION.+ --> SIGN TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION.- --> SIGN TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION.; --> synch
SIMPLE_EXPRESSION.id --> TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION.num --> TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION.relop --> synch
SIMPLE_EXPRESSION1.) --> \L
SIMPLE_EXPRESSION1.; --> \L
SIMPLE_EXPRESSION1.addop --> 'addop' TERM SIMPLE_EXPRESSION1
SIMPLE_EXPRESSION1.relop --> \L
STATEMENT.$ --> synch
STATEMENT.float --> DECLARATION
STATEMENT.id --> ASSIGNMENT
STATEMENT.if --> IF
STATEMENT.int --> DECLARATION
STATEMENT.while --> WHILE
STATEMENT.} --> synch
STATEMENT_LIST.$ --> synch
STATEMENT_LIST.float --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST.id --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST.if --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST.int --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST.while --> STATEMENT STATEMENT_LIST1
```

```
STATEMENT_LIST1.$ --> \L
STATEMENT_LIST1.float --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST1.id --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST1.if --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST1.int --> STATEMENT STATEMENT_LIST1
STATEMENT_LIST1.while --> STATEMENT STATEMENT_LIST1
TERM.( --> FACTOR TERM1
TERM.) --> synch
TERM.; --> synch
TERM.addop --> synch
TERM.id --> FACTOR TERM1
TERM.num --> FACTOR TERM1
TERM.relop --> synch
TERM1.) --> \L
TERM1.; --> \L
TERM1.addop --> \L
TERM1.mulop --> 'mulop' FACTOR TERM1
TERM1.relop --> \L
WHILE.$ --> synch
WHILE.float --> synch
WHILE.id --> synch
WHILE.if --> synch
WHILE.int --> synch
WHILE.while --> 'while' '(' EXPRESSION ')' '{' STATEMENT '}'
WHILE.} --> synch
```

Conclusions

Phase 1

Developing a lexical analyzer doesn't depend on a specific language, what is only needed is the rules of the language from which an NFA diagram can be built.

Phase 2

Developing a lexical parser doesn't depend on a specific language, what is only needed is the production of the language from which the parsing table can be constructed and using the lexemes from the lexical analyzer the input of a language can be analyzed to know if it follows the language rules.

Phase 3

Applying semantic rules can be integrated in the parsing phase through applying semantic rules within the productions to generate an intermediate language which in our case is the Java bytecode.

Comments about used tools

Phase 1

Flex for simulating lexical analyzer that is used for the Bonus part.

Phase 3

Flex and Bison are tools for building programs that handle structured input. They were originally tools for building compilers, but they have proven to be useful in many other areas.

Assumptions

Phase 1

Reading the file of tokens with specific assumptions for parsing.
Rules reserved symbols (+,*,=,(,),{,}) preceded by a \ character.
Epsilon represented by \L

Phase 2

Reading the file of productions with specific assumptions for parsing.
Terminals surrounded by single quotes.
Epsilon represented by \L

Phase 3

Incrementing the counter in For loop must be assignment,
i = i + 1; instead of i++.

References

Phase 1

https://www.tutorialspoint.com/automata_theory/dfa_minimization.htm

https://www.cs.odu.edu/~toida/nerzic/390teched/regular/fa/nfa-lambda-definitions.html

Phase 2

https://www.codeproject.com/Articles/5162249/How-to-Make-an-LL-1-Parser-Lesson-1#_articleTop
https://people.cs.pitt.edu/~jmisurda/teaching/cs1622/handouts/cs1622-first_and_follow.pdf

Phase 3

http://www.cs.cmu.edu/~wmh/bison.html

https://www.youtube.com/watch?v=RvYA12UiTFc

https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html

http://tldp.org/HOWTO/Lex-YACC-HOWTO-6.html

https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html

Role of each student

ElSayed AbdelNasser ElSayed (14)

Phase 1: **used the minimized DFA to build the simulation class.**

Phase 2: **building the parsing table.**

Phase 3: **Implementation lexical.l & semantics.y classes.**

Ziad Hisham Ali (21)

Phase 1: **Implemented the Rules class and its connection with the NFA to give it the inputs it needs.**

Phase 2: **Implemented the Follow class responsible for computing the follow sets of the non-terminals.**

Phase 3: **Implementation lexical.l & semantics.y classes.**

Karim Atef Ahmed (33)

Phase 1: **Implementation of NFA classes.**

Phase 2: **Implementation of LL1Grammar & First classes**

Phase 3: **Implementation of lexical.l & semantics.y classes.**

Youssef Ahmed Sayed (64)

Phase 1: **Building DFA transition table and implementing it’s minimization.**

Phase 2: **Reading the file of productions and parsing it.**

Phase 3**: Implementation lexical.l & semantics.y classes.**