Programming Languages Translation

Phase 2 : Parser generator

Name	ID
Mohamed Harraz	4608
Sherif Rafik	4635
Omar Nasr	4730

Project Objective:

The aim of this phase is to practice techniques for building automatic parser generation tools.

Project description:

The parser generator expects an LL(1) grammar as input, it computes the first and follow and uses them to construct the predictive parsing table.

The table is used to derive a predictive top-down parser. If the input grammar is not LL(1), an appropriate error message should be produced.

If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing

We've combined the lexical analyzer from phase 1 and the parser from this project such that the lexical analyzer is to be called to get the next token.

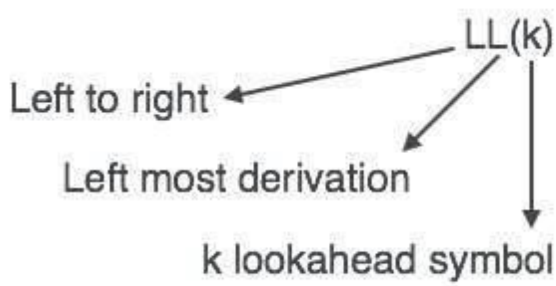
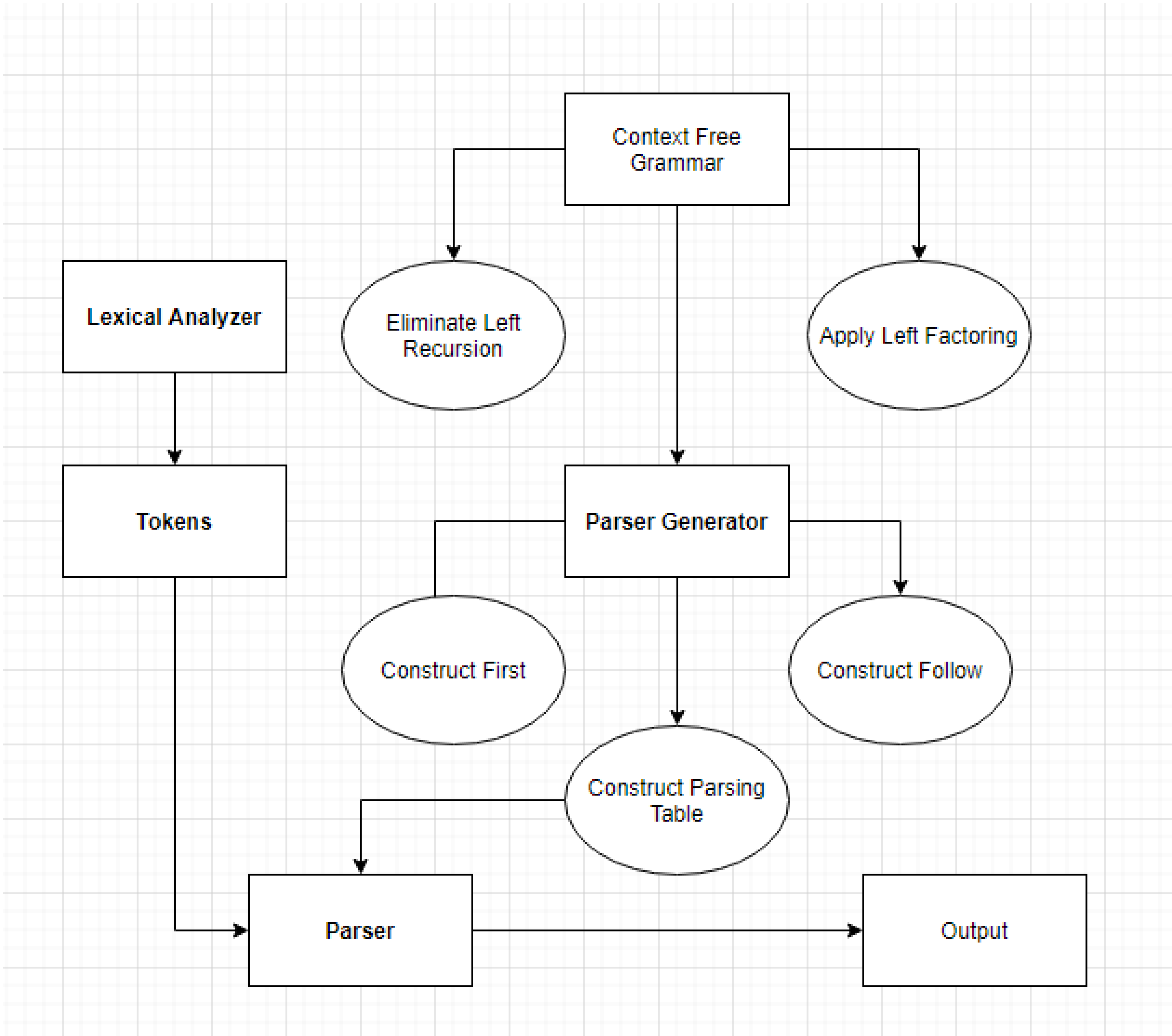


Figure 1: Top-down parser, K = 1 in our case

Project flow:



Data Structures:

HashMaps:

A Java built-in data structure that stores data in (key, value) pairs. To access a value, one must know its key.

ArrayLists:

A Java built-in data structure that presents a dynamic array.

Stack:

A Java built-in data structure that's based on the LIFO (last-in-first-out) principle.

- Used in the parser stack algorithm

HashSets:

A Java built-in data structure that represents a set, where the input is hashed before insertion.

- Used in the construction of the **first** and **follow**

Pair:

A Java built-in data structure that stores data in (key, value) manner.

Trie:

A data structure which is used to store the collection of strings and makes searching of a pattern in words easier.

- Used in applying **left factoring** (if needed)

Algorithms and techniques:

Left Recursion Elimination:

The problem with left recursion is if it is present in any grammar, during parsing the syntax there's a chance that the grammar will create an infinite loop. This is because at every time of production of grammar S will produce another S without checking any condition.

We will eliminate both indirect left recursion and direct (immediate) left recursion

Code snippet:

```
private void eliminateLeftRecursion() {
    eliminateIndirectLeftRecursion();
    eliminateImmediateLeftRecursion();
}
private void eliminateIndirectLeftRecursion() {
    //first we have some ordering of nonterminals
    //replace all nonterminals that precede me with their production
    //so i can easily remove the immediate left recursion later
    ArrayList<String> nontermnials = rulesCont.getProductionRules();
    for (int i = 0; i < nontermnials.size(); i++) {
        String currentNonterminal = nontermnials.get(i); // Ai
        for (int j = 0; j < i; j++) {
            String subNonterminal = nontermnials.get(j);// Aj
            // for each Ai -> Aj b replace Aj by Ai -> a b | b b | c b as Aj -> a | b | c
            replaceNonterminal(currentNonterminal, subNonterminal);
        }
    }
}
private void replaceNonterminal(String nonterminal, String subNonterminal) {
    //helper function takes care of this " each Ai -> Aj b replace Aj by Ai -> a b | b b |
c b as Aj -> a | b | c "
    ArrayList<ArrayList<String>> modifiedProduction = new ArrayList<>();
    ArrayList<ArrayList<String>> nonterminalProductions =
this.rulesCont.getProductionRule(nonterminal);
    ArrayList<ArrayList<String>> subNonterminalProductions =
this.rulesCont.getProductionRule(subNonterminal);
    for (ArrayList<String> nonterminalProduction : nonterminalProductions) {
        if (subNonterminal.equals(nonterminalProduction.get(0))) {
            ArrayList<ArrayList<String>> currentProductions = new ArrayList<>();
            nonterminalProduction.remove(0);
            for (ArrayList<String> subNonterminalProduction : subNonterminalProductions) {
                ArrayList<String> currentProduction = new ArrayList<>();
                currentProduction.addAll(subNonterminalProduction);
                currentProduction.addAll(nonterminalProduction);
                currentProductions.add(currentProduction);
            }
            modifiedProduction.addAll(currentProductions);
        } else {
            ArrayList<String> currentProduction = new ArrayList<>();
            currentProduction.addAll(nonterminalProduction);
            modifiedProduction.add(currentProduction);
        }
    }
    this.rulesCont.changeProductionEntry(nonterminal, modifiedProduction);

}
```

Left Factoring:

Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracing by the parser.

Code snippet:

```
private void leftFactoring() {
    //left factoring elimination that relies on using Trie data structure
    // for each nonterminal i make a trie out of it's production
    // the trie has a hashmap<key:string,value:pair<key:next node,value: frequency of that
string>>
    //if frequency of that string in the trie more than 1 then i need to make new non
terminal and productions to it
    // then i recurse again on the new nonterminal until no more new nonterminals are
needed
    ArrayList<String> nontermnials = rulesCont.getProductionRules();
    for (int i = 0; i < nontermnials.size(); i++) {
        String nonterminal = nontermnials.get(i);
        TrieNode root = new TrieNode();
        ArrayList<ArrayList<String>> productions =
rulesCont.getProductionRule(nonterminal);
        for (ArrayList<String> production : productions) {
            root.addString(production, 0);
        }
        constructLeftFactoredRule(root, nonterminal);

    }

}
```

```
private void constructLeftFactoredRule(TrieNode currentNode, String nonterminal) {
    ArrayList<String> currentWords = currentNode.getNodeKeys();
    for (String word : currentWords) {
        Pair<TrieNode, Integer> nextNode = currentNode.getNode(word);
        if (nextNode.getValue() > 1) {
            String newNonTerminal = getNewNonTerminal(nonterminal,
Constant.NONTERMINAL_LEFT_FACTOR_DASH);
            String newRule = newNonTerminal + Constant.PRODUCTION_RULE_ASSIGNMENT;
            ArrayList<ArrayList<String>> productions =
rulesCont.getProductionRule(nonterminal);
            for (ArrayList<String> production : productions) {
                if (production.get(0).equals(word)) {
                    // if rule is completely factored and nothing left then i need to add
epsilon
                    if (production.size() == 1) {
                        newRule += Constant.EPSILON;
                    } else {
                        for (int i = 1; i < production.size(); i++) {
                            newRule += production.get(i) + " ";
                        }
                    }
                    newRule += Constant.OR;
                    production.clear();
                    production.add(word);
                    production.add(newNonTerminal);
                }
            }
            // Remove duplicates after left factoring
            for (int i = 0; i < productions.size(); i++) {
                ArrayList<String> production = productions.get(i);
                for (int j = i + 1; j < productions.size(); j++) {
                    ArrayList<String> otherProduction = productions.get(i);
                    if (production.equals(otherProduction)) {
                        productions.remove(j);
                    }
                    j--;
                }
            }
            newRule = newRule.substring(0, newRule.length() - 1);// remove last '|'
            ParserLineProcessor.getInstance().processLine(newRule, rulesCont);
            constructLeftFactoredRule(nextNode.getKey(), newNonTerminal);
        }
    }
}
```

First:

Rules to compute FIRST set:

1- If x is a terminal, then FIRST(x) = {'x'}
2- If x-> Є, is a production rule, then add Є to FIRST(x).
3- If X->Y1 Y2 Y3…. Yn is a production,
   a. FIRST(X) = FIRST(Y1)
   b. If FIRST(Y1) contains Є then FIRST(X) = {FIRST(Y1) – Є} U {FIRST(Y2)}
   c. If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Code snippet:

```
private void first() {
    ArrayList<String> visited = new ArrayList<String>();

    for (int i = 0; i < nonTerminals.size(); i++) {
        String nonTerminal = nonTerminals.get(i);
        if (!visited.contains(nonTerminal)) {
            visited.add(nonTerminal);
            firstRecursive(nonTerminal, visited);
        }
    }
    return;
}
private void firstRecursive(String start, ArrayList<String> visited) {
    if (!grammar.isNonTerminal(start) || start.equals(Constant.EPSILON)) {
        first.put(start, new HashSet<String>());
        first.get(start).add(start);
        return;
    }
    // Get the right hand side of the current non terminal
    ArrayList<ArrayList<String>> rhs = grammar.getRHS(start);
    // Loop over the rhs
    for (ArrayList<String> temp : rhs) {
        // Get the first word
        for (int j = 0; j < temp.size(); j++) {
            String word = temp.get(j);
            if (!visited.contains(word)) {
                visited.add(word);
                // Make recursive call
                firstRecursive(word, visited);
            }
            // Get the first of the child
            HashSet<String> firstOfChild = first.get(word);
            // Add it to the parent's hashset
            for (String s : firstOfChild) {
                first.get(start).add(s);
            }
            // If the first of the word contains epsilon, then stop
            if (!first.get(word).contains(Constant.EPSILON)) {
                if (first.get(start).contains(Constant.EPSILON) && first.get(start).size() >
2) {
                    first.get(start).remove(Constant.EPSILON);
                }
                break;
            }
        }
    }
}
```

Follow:

Rules to compute FIRST set:

1- FOLLOW (Starting Symbol) = {$}
2- If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST(q) except Є is in FOLLOW(B).
3- If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW(B).
4- If A->pBq is a production and FIRST(q) contains Є, then FOLLOW(B) contains {FIRST(q) – Є} U FOLLOW(A)

Code snippet:

```
private void follow() {
    HashMap<String, ArrayList<String>> followDependency = new LinkedHashMap<String,
ArrayList<String>>();
    for (int i = 0; i < nonTerminals.size(); i++) {
        followDependency.put(nonTerminals.get(i), new ArrayList<>());
    }

    // First rule : Add to the follow Of starting non terminal = $
    follow.get(grammar.getStartingNonTerminal()).add(Constant.END_MARKER);

    // Loop over all the non terminals
    for (int i = 0; i < nonTerminals.size(); i++) {
        String nonTerminal = nonTerminals.get(i);
        // Get the RHS
        ArrayList<ArrayList<String>> rhs = grammar.getRHS(nonTerminal);
        for (ArrayList<String> temp : rhs) {
            for (int j = 0; j < temp.size(); j++) {
                String current = temp.get(j);
                // If its a non terminal
                if (grammar.isNonTerminal(current)) {
                    /*
                     * If J == last Index then the follow of the current depends on the
follow of
                     * the original non terminal
                     */
                    if (j == temp.size() - 1) {
                        followDependency.get(current).add(nonTerminal);
                    } else {
                        // Loop over the rest of the rhs
                        for (int k = j + 1; k < temp.size(); k++) {
                            String following = temp.get(k);
                            // If the next is a non terminal
                            if (grammar.isNonTerminal(following)) {
                                /*
                                 * If k == Last index and there's an epsilon in its start
Then the follow of the
                                 * current depends on the follow of the original non
terminal
                                 */
                                if (k == temp.size() - 1 &&
first.get(following).contains(Constant.EPSILON))
                                    followDependency.get(current).add(nonTerminal);
```

```
                                // Follow of current = first of next except epsilon
                                for (String first : first.get(following)) {
                                    this.follow.get(current).add(first);
                                }
                                // If you encounter an epsilon in its start then continue
looping
                                if (first.get(following).contains(Constant.EPSILON))
                                    follow.get(current).remove(Constant.EPSILON);
                                    // Else break the loop and DONE
                                else
                                    break;
                            } else {
                                this.follow.get(current).add(following);
                                break;
                            }
                        }
                    }
                }
            }
        }
    }
    // Loop until no updates are done
    while (true) {
        boolean updates = false;
        // Loop over all the non terminals
        for (Entry<String, ArrayList<String>> entry : followDependency.entrySet()) {
            // Get the dependency of each non terminal
            String nonTerminal = entry.getKey();
            ArrayList<String> dependency = entry.getValue();
            // Save old follow of the current non terminal
            HashSet<String> oldFollow = follow.get(nonTerminal);
            // Loop over the current non terminals dependency and update the follow
            for (int i = 0; i < dependency.size(); i++) {
                for (String s : follow.get(dependency.get(i)))
                    follow.get(nonTerminal).add(s);
                // If the old follow is not the same as the updated follow, set the flag
                if (!follow.get(nonTerminal).equals(oldFollow)) {
                    updates = true;
                }
            }
        }
        // If no more updates are done, terminate loop
        if (!updates)
            break;
    }
}
```

Parsing table construction:

Now, after computing the First and Follow set for each *Non-Terminal symbol* we have to construct the Parsing table. In the table Rows will contain the Non-Terminals and the column will contain the Terminal Symbols.

All the Null Productions of the Grammars will go under the Follow elements and the remaining productions will lie under the elements of First set.

Code snippet:

```
private void buildTable() { /** loop for all non terminals to fill the table */
    for (String nonTerminalEntry : nonTerminals) {
        boolean hasEpsilon = false;
        /** loop through all first(curr non terminal) find it's production rule entry
*/
        for (String firstEntry : first.get(nonTerminalEntry)) {
            if (firstEntry.equals(Constant.EPSILON)) {
                hasEpsilon = true;
                continue;
            }
            /** loop through all RHS rules for curr(non terminal) */
            for (ArrayList<String> productionRule : grammar.getRHS(nonTerminalEntry)) {
                /** check if it contains curr first entry add it in table */
                if (first.get(productionRule.get(0)).contains(firstEntry)) {
                    /**
                     * if the entry being filled for terminal input char already filled
report an
                     * ambiguity error and fill it again
                     */
                    if (parsingTable.get(nonTerminalEntry).containsKey(firstEntry)) {

parsingTable.get(nonTerminalEntry).get(firstEntry).add(productionRule);
                        isAmbiguousGrammar = true;
                    } else { /** create a new entry in table and fill it */
                        ArrayList<ArrayList<String>> productionRulesEntry = new
ArrayList<>();
                        productionRulesEntry.add(productionRule);
                        parsingTable.get(nonTerminalEntry).put(firstEntry,
productionRulesEntry);
                    }
                }
                /**
                 * looped for all first of the non terminal entry to fill it with a
production
                 * rule
                 */
            }
        }
        /**
         * loop through all follow(curr non terminal) and fill it with epsilon or sync
         */
        for (String followEntry : follow.get(nonTerminalEntry)) {
            if (hasEpsilon) { /** if first has epsilon [not terminal,follow] = epsilon
*/
                ArrayList<String> epsilonRule = new ArrayList<>();
                epsilonRule.add(Constant.EPSILON);
```

```
                /** if entry already exists report ambiguity error and refill it */
                if (parsingTable.get(nonTerminalEntry).containsKey(followEntry)) { /**/
                    isAmbiguousGrammar = true;

parsingTable.get(nonTerminalEntry).get(followEntry).add(epsilonRule);
                } else {
                    ArrayList<ArrayList<String>> productionRulesEntry = new
ArrayList<>();
                    productionRulesEntry.add(epsilonRule);
                    parsingTable.get(nonTerminalEntry).put(followEntry,
productionRulesEntry);
                }
            } else if (!first.get(nonTerminalEntry).contains(followEntry)) { /** if
first doesn't has epsilon [non terminal,follow] = epsilon */
                ArrayList<String> syncRule = new ArrayList<>();
                syncRule.add(Constant.SYNC_TOK);
                /** if entry already exists report ambiguity error and refill it */
                if (parsingTable.get(nonTerminalEntry).containsKey(followEntry)) {
                    isAmbiguousGrammar = true;
                    parsingTable.get(nonTerminalEntry).get(followEntry).add(syncRule);
                } else { /** create new entry in table */
                    ArrayList<ArrayList<String>> productionRulesEntry = new
ArrayList<>();
                    productionRulesEntry.add(syncRule);
                    parsingTable.get(nonTerminalEntry).put(followEntry,
productionRulesEntry);
                }
            }
        }
    }
}
```

Parse:

Code snippet:

```
public void parse() {
    int inputTokenIndex = 0;
    Stack<String> stack = new Stack<>();
    stack.push(Constant.END_MARKER);
    stack.push(grammar.getStartingNonTerminal());
    while (!stack.empty() && inputTokenIndex != inputTokens.size()) {

        Pair<String, Pair<String, String>> logEntry;

        StringBuilder stackContent = new StringBuilder();
        for (String string : stack) {
            stackContent.append(string + " ");
        }

        StringBuilder inputContents = new StringBuilder();
        for (int j = inputTokenIndex; j < inputTokens.size(); j++) {
            inputContents.append(inputTokens.get(j) + " ");
        }


        String TOS = stack.pop();

        if (grammar.isNonTerminal(TOS)) { /** if top of stack is non terminal */
            /** if top of stack leads to empty entry */
            if
(!parsingTable.get(TOS).containsKey(inputTokens.get(inputTokenIndex))) {
                logEntry = new Pair(stackContent.toString(), new
Pair<>(inputContents.toString(),
                        "empty entry action: skip this token \'" +
inputTokens.get(inputTokenIndex) + "\'"));
                inputTokenIndex++;
                stack.push(TOS);
            }
            /** if top of stack leads to epsilon */
            else if
(parsingTable.get(TOS).get(inputTokens.get(inputTokenIndex)).get(0).get(0).equal
s(Constant.EPSILON)) {
                logEntry = new Pair(stackContent.toString(), new
Pair<>(inputContents.toString(),
                        "epsilon action: pop stack \'" + TOS + "\'"));
            }
```

```
            /** if top of stack is SYNC_TOK */
            else if
(parsingTable.get(TOS).get(inputTokens.get(inputTokenIndex)).get(0).get(0).equal
s(Constant.SYNC_TOK)) {
                logEntry = new Pair(stackContent.toString(), new
Pair<>(inputContents.toString(),
                        "SYNC action: pop stack \'" + TOS + "\'"));
            } else {/** a production rule needs to be pushed to stack */
                int lengthOfArray =
parsingTable.get(TOS).get(inputTokens.get(inputTokenIndex)).get(0).size();
                for (int i = lengthOfArray - 1; i >= 0; i--) {

stack.push(parsingTable.get(TOS).get(inputTokens.get(inputTokenIndex)).get(0).ge
t(i));
                }
                logEntry = new Pair(stackContent.toString(), new
Pair<>(inputContents.toString(),
                        "production rule pushed to stack"));
            }
        } else { /** if top of stack is terminal */
            String actionLog;
            /** if input token match top of stack */
            if (TOS.equals(inputTokens.get(inputTokenIndex))) {
                actionLog = "match action: skip this token \'" +
inputTokens.get(inputTokenIndex) + "\'";
            }
            /** if input token doesn't match top of stack */
            else {
                actionLog = "no match action: skip this token \'" +
inputTokens.get(inputTokenIndex) + "\'";
                stack.push(TOS);
            }
            logEntry = new Pair(stackContent.toString(), new
Pair<>(inputContents.toString(), actionLog));
            inputTokenIndex++;
        }
        log.add(logEntry);
    }
}
```

Assumptions:

- The concatenation symbol is **"`"**
- The end marker is the dollar sign **"$"**
- On Eliminating left recursion, we add **"_DASH"** to the non-terminal
- On Applying left factoring, we add **"_HASH"** to the non-terminal
- Synchronization token is **"SYNC"**

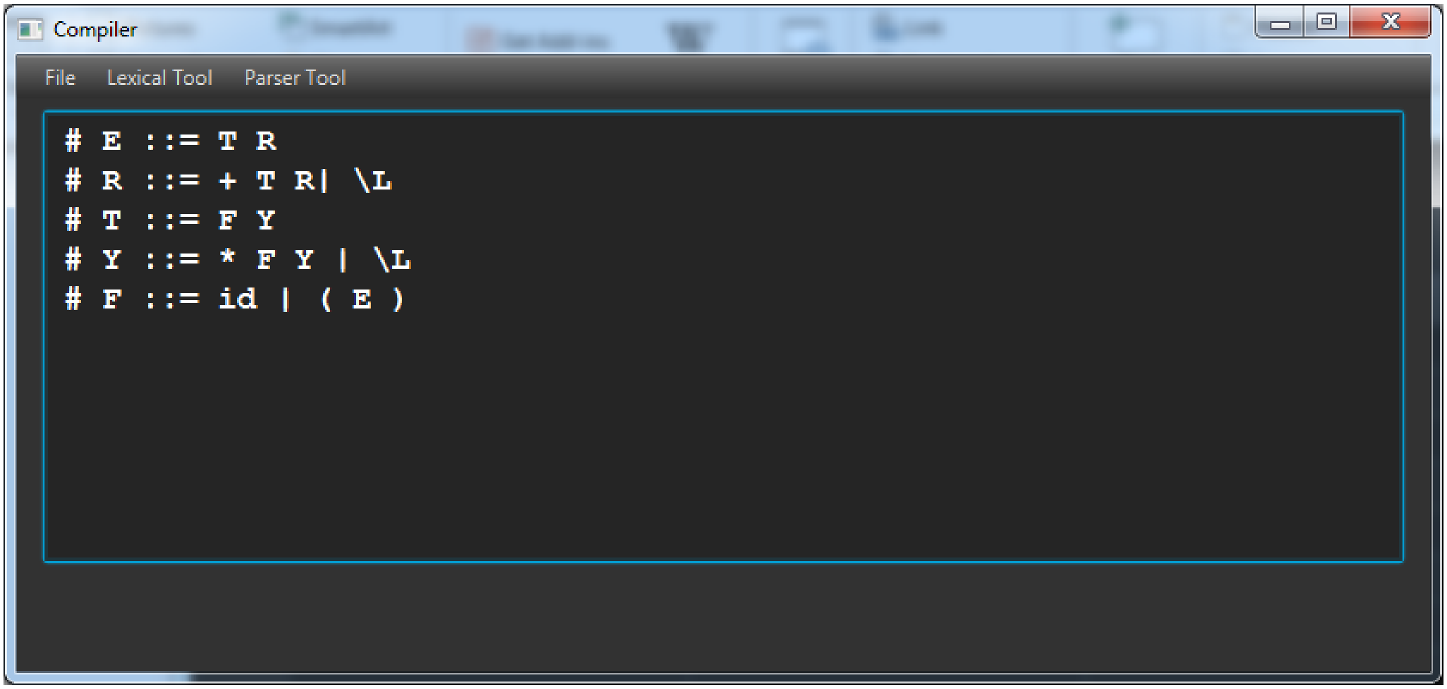
Project structure:

Project follows the MVC (model-view-controller) design pattern.

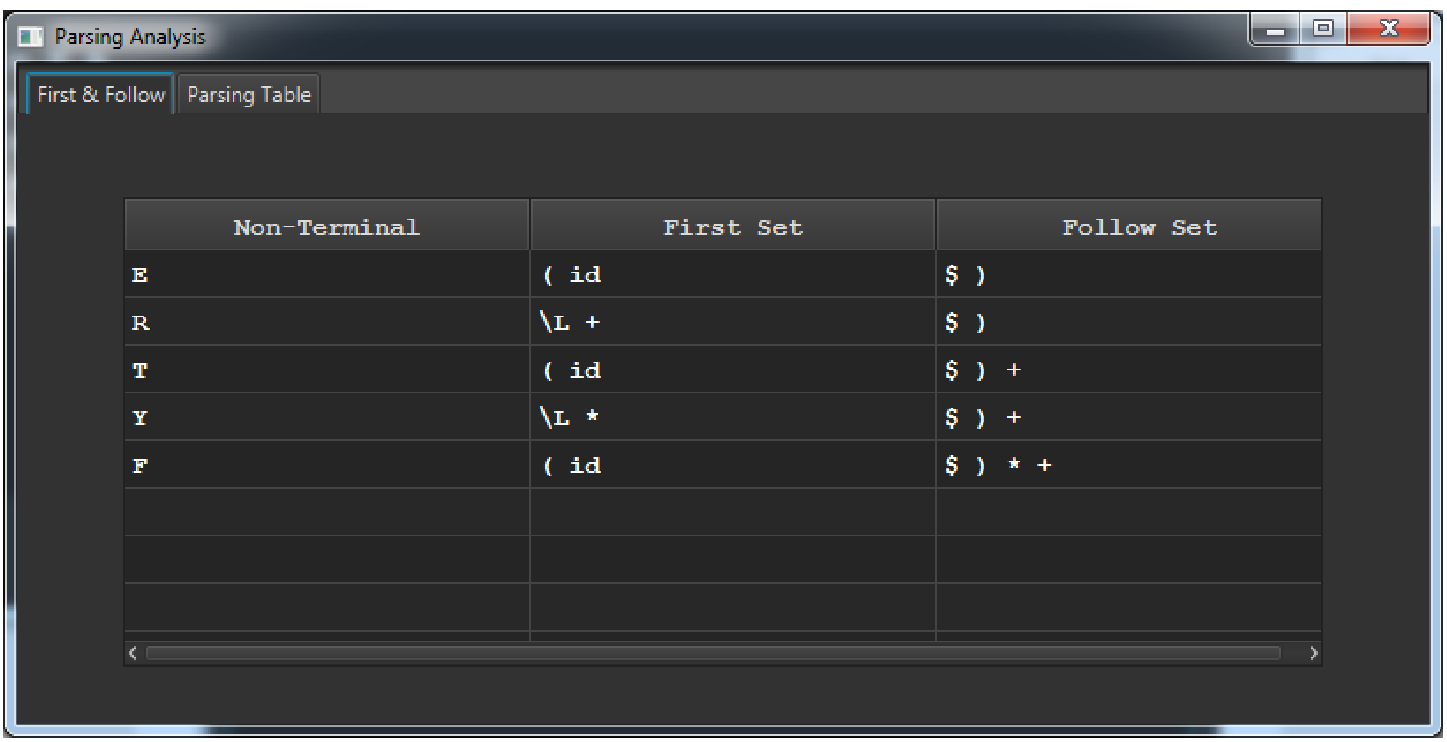
1- Every GUI related class is found in the view package
2- Every Lexical analyzer related class is found in the model.lexical package
    a. Construction
        i. Line Processor class
        ii. Rules Container class
    b. DFA
        i. DFA class
        ii. DFA Optimizer class
    c. Graph
        i. Graph class
        ii. Node class
    d. NFA
        i. Keyword class
        ii. NFA class
        iii. Punctuation class
        iv. Regular Definition class
        v. Regular Expression class
3- Every parser related class is found in the model.parser package
    a. Construction
        i. Parser line processor class
        ii. Parser rules container class
    b. CFG
        i. CFG class (Context free grammar)
    c. Parser
        i. Parser class
        ii. Parser Generator class
4- Controller class is found in the controller package
5- Helper classes are found in the utilities package

Sample runs:

Context free grammar:

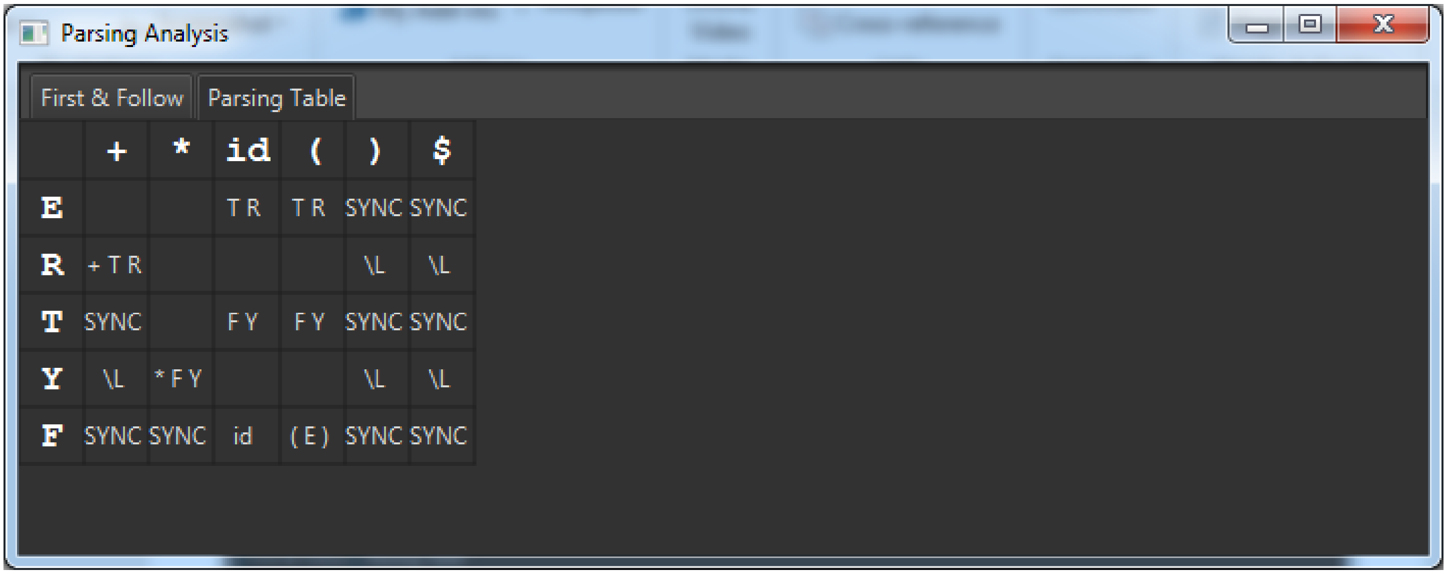


First and follow:



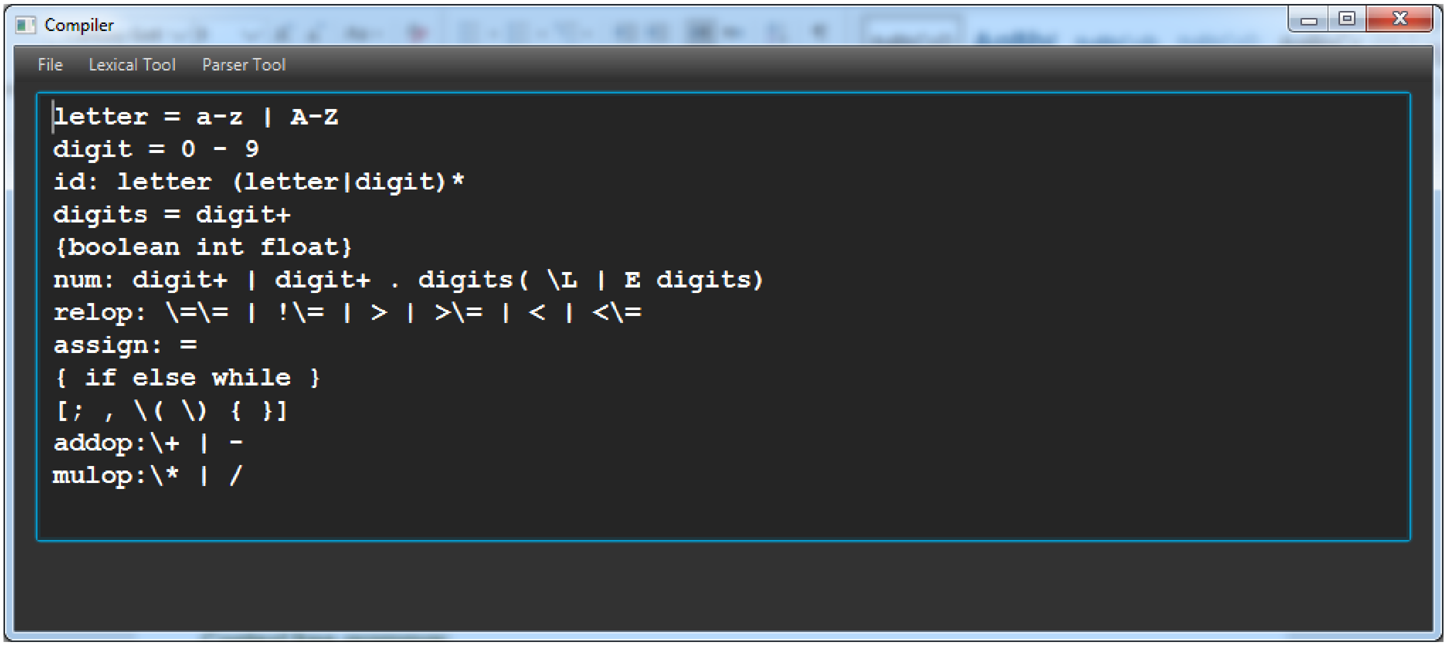
Non-Terminal	First Set	Follow Set
E	(id	$)
R	\L +	$)
T	(id	$) +
Y	\L *	$) +
F	(id	$) * +

Parsing table:

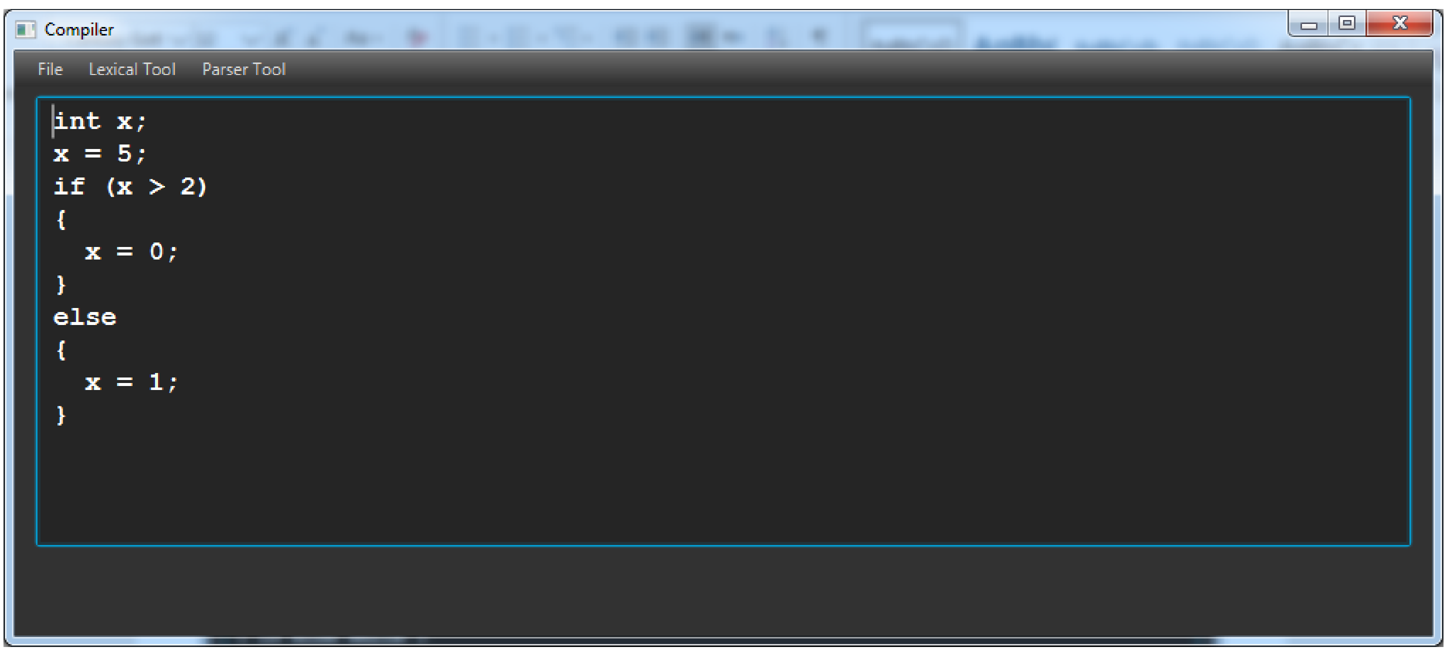


	+	*	id	(	)	$
E			T R	T R	SYNC	SYNC
R	+ T R				\L	\L
T	SYNC		F Y	F Y	SYNC	SYNC
Y	\L	* F Y			\L	\L
F	SYNC	SYNC	id	(E)	SYNC	SYNC

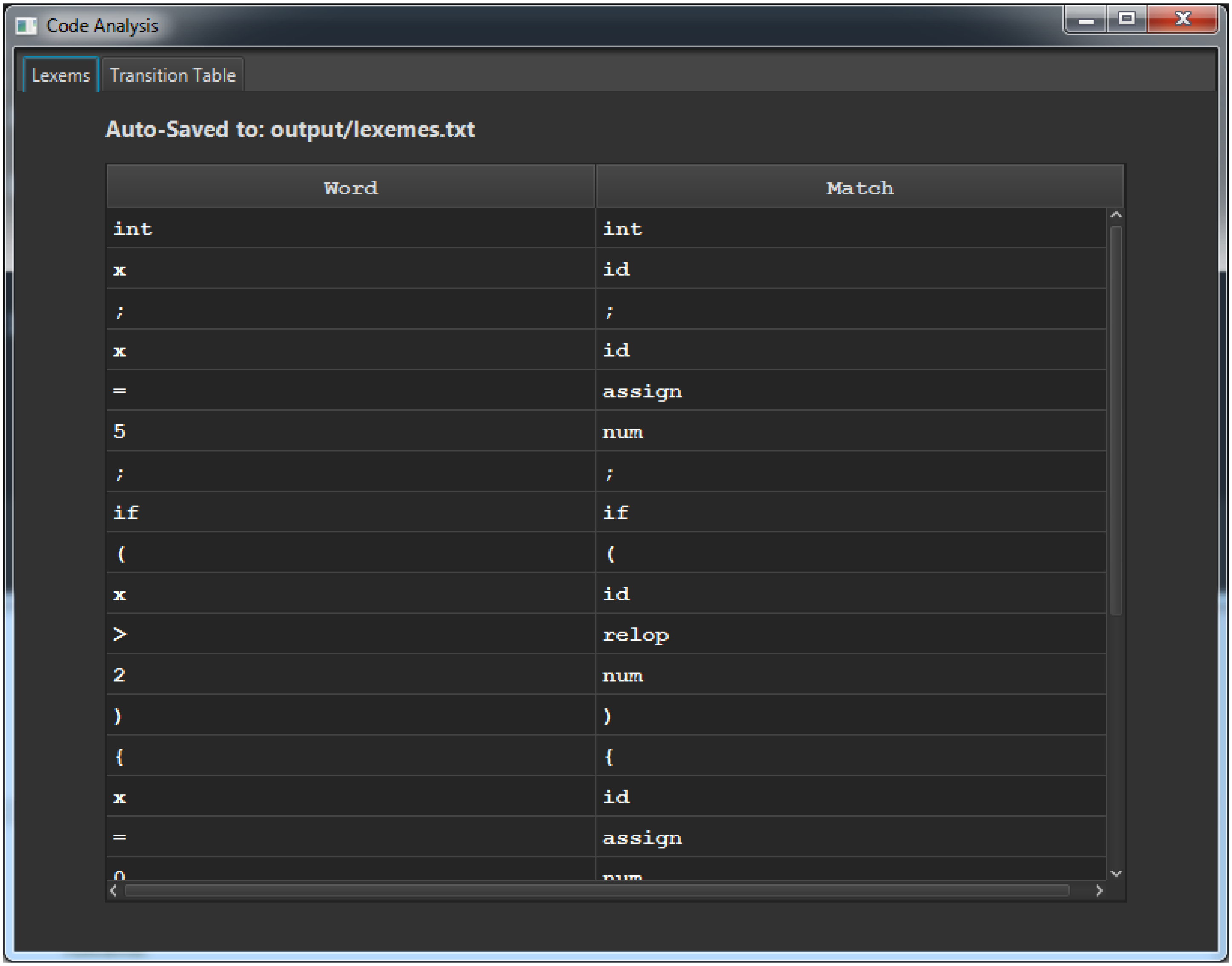
Lexical Rules:



Program:

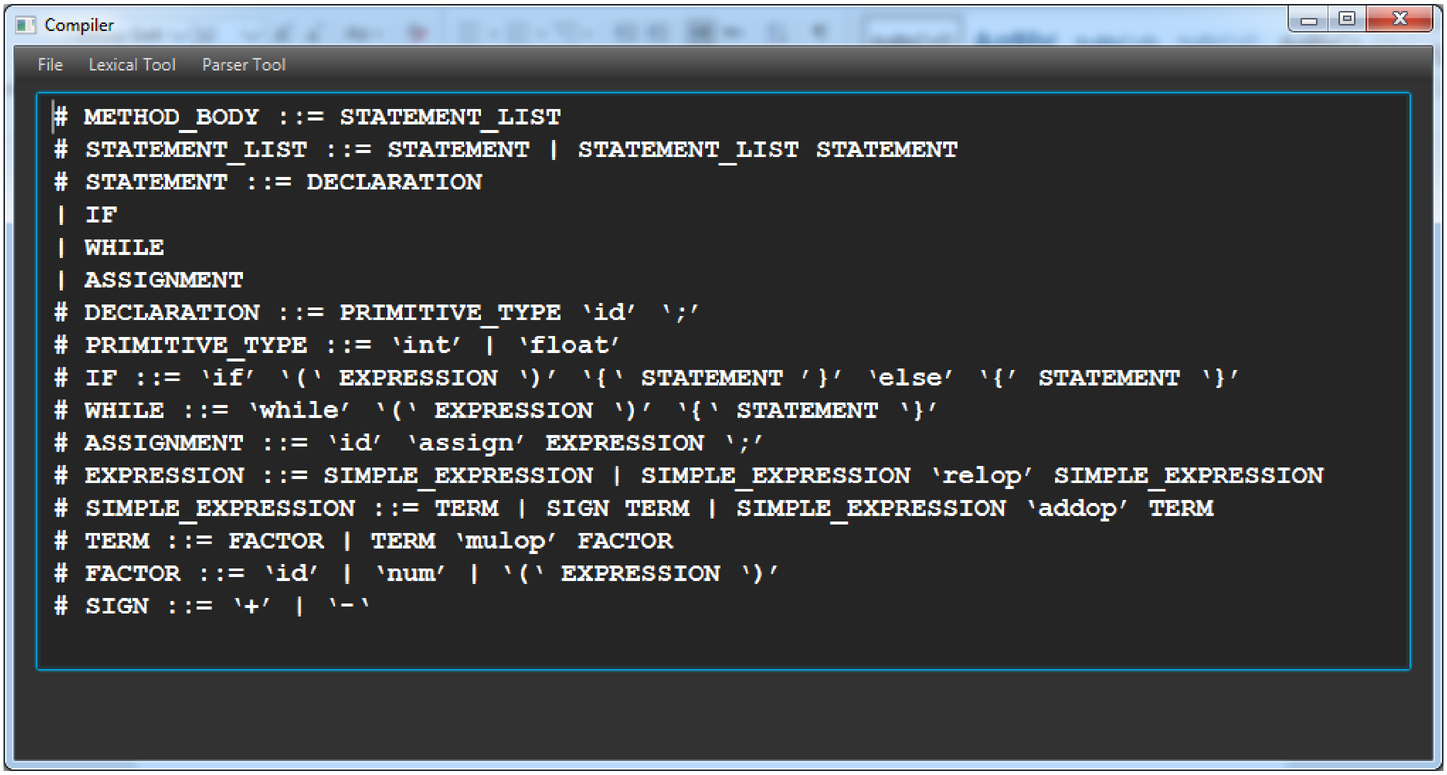


Tokens:



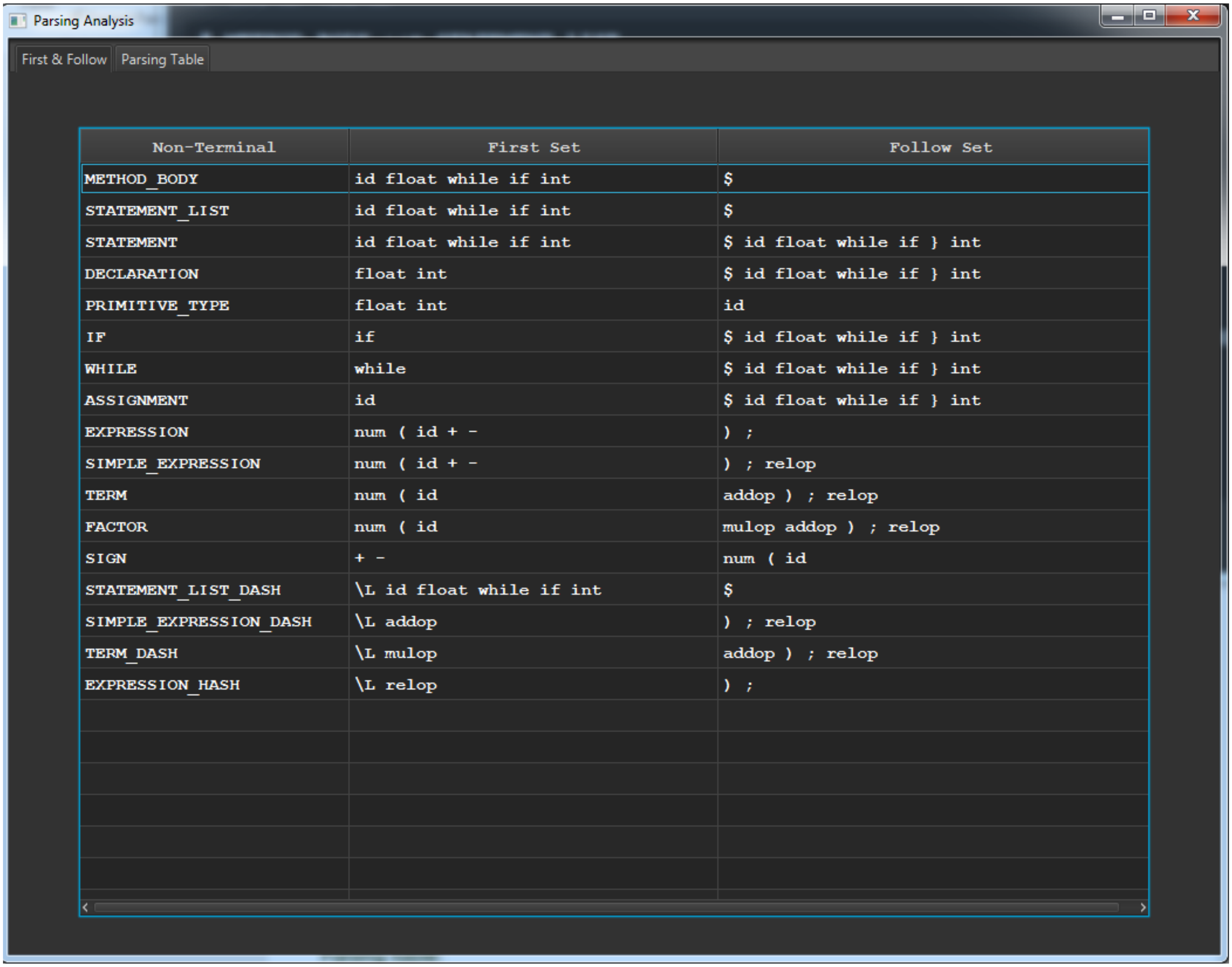
Word	Match
int	int
x	id
;	;
x	id
=	assign
5	num
;	;
if	if
(	(
x	id
>	relop
2	num
)	)
{	{
x	id
=	assign

Context free grammar:



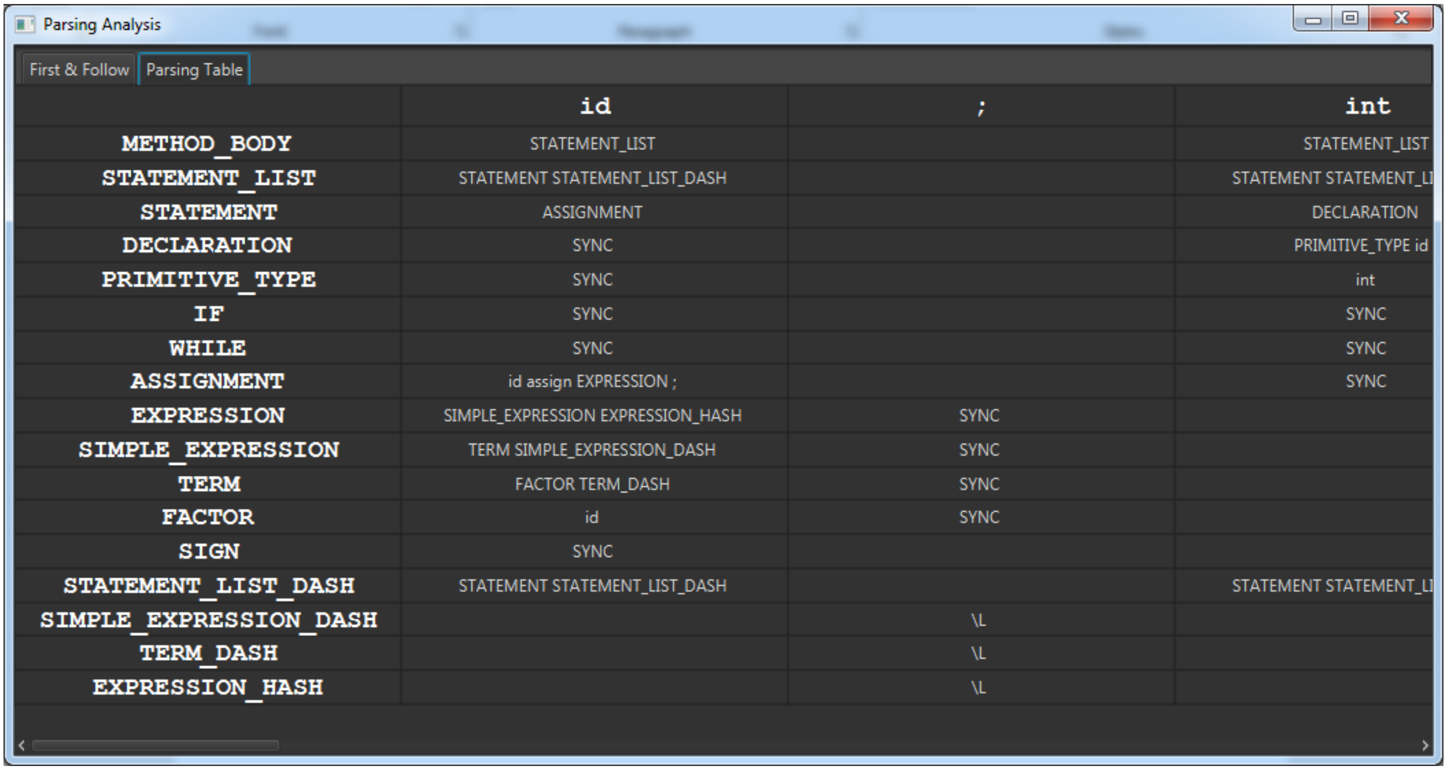

```
# METHOD_BODY ::= STATEMENT_LIST
# STATEMENT_LIST ::= STATEMENT | STATEMENT_LIST STATEMENT
# STATEMENT ::= DECLARATION
| IF
| WHILE
| ASSIGNMENT
# DECLARATION ::= PRIMITIVE_TYPE 'id' ';'
# PRIMITIVE_TYPE ::= 'int' | 'float'
# IF ::= 'if' '(' EXPRESSION ')' '{' STATEMENT '}' 'else' '{' STATEMENT '}'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

First and follow:



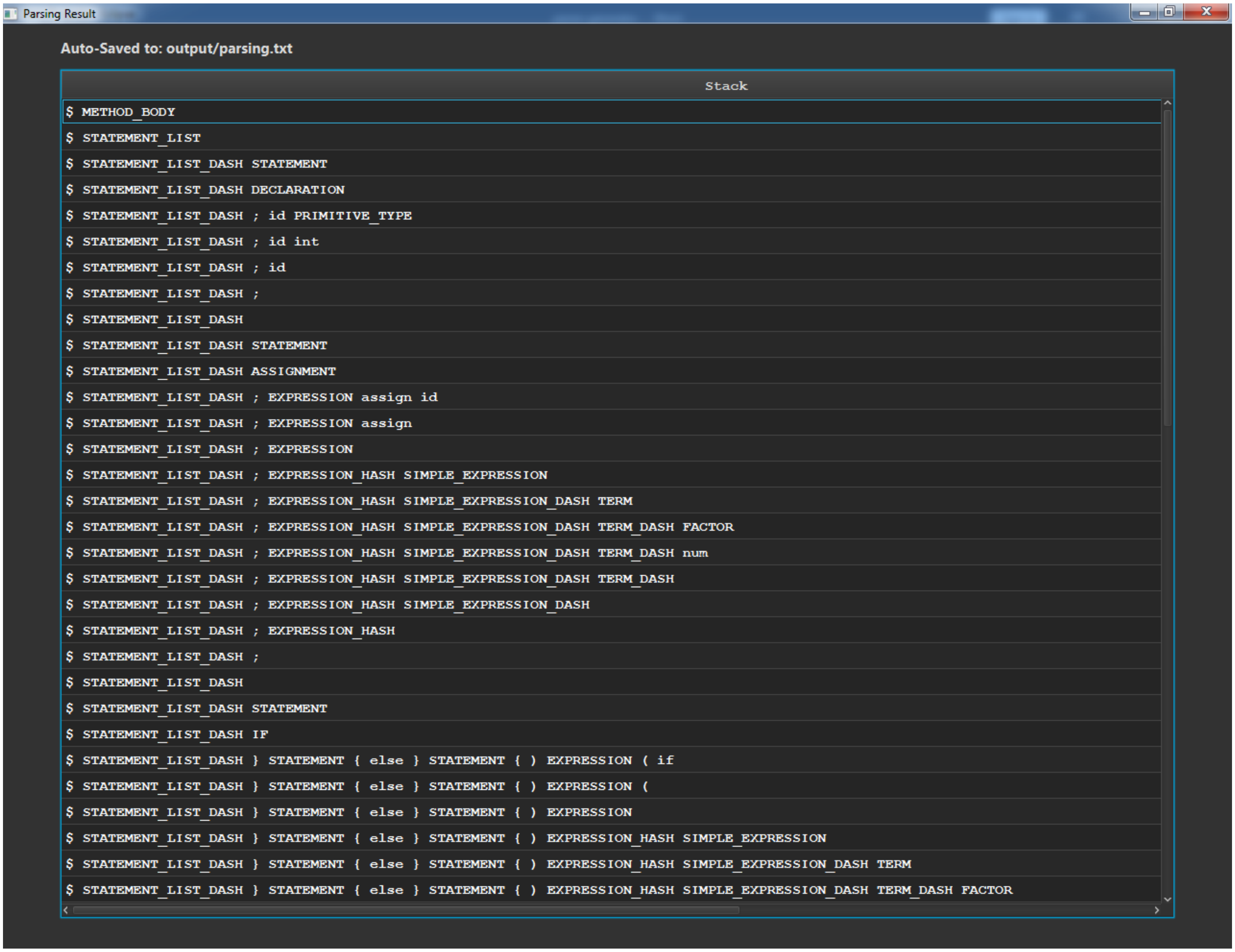
Non-Terminal	First Set	Follow Set
METHOD_BODY	id float while if int	$
STATEMENT_LIST	id float while if int	$
STATEMENT	id float while if int	$ id float while if } int
DECLARATION	float int	$ id float while if } int
PRIMITIVE_TYPE	float int	id
IF	if	$ id float while if } int
WHILE	while	$ id float while if } int
ASSIGNMENT	id	$ id float while if } int
EXPRESSION	num (id + -	) ;
SIMPLE_EXPRESSION	num (id + -	) ; relop
TERM	num (id	addop) ; relop
FACTOR	num (id	mulop addop) ; relop
SIGN	+ -	num (id
STATEMENT_LIST_DASH	\L id float while if int	$
SIMPLE_EXPRESSION_DASH	\L addop	) ; relop
TERM_DASH	\L mulop	addop) ; relop
EXPRESSION_HASH	\L relop	) ;

Parsing table:



	id	;	int
METHOD_BODY	STATEMENT_LIST		STATEMENT_LIST
STATEMENT_LIST	STATEMENT STATEMENT_LIST_DASH		STATEMENT STATEMENT_LI
STATEMENT	ASSIGNMENT		DECLARATION
DECLARATION	SYNC		PRIMITIVE_TYPE id
PRIMITIVE_TYPE	SYNC		int
IF	SYNC		SYNC
WHILE	SYNC		SYNC
ASSIGNMENT	id assign EXPRESSION ;		SYNC
EXPRESSION	SIMPLE_EXPRESSION EXPRESSION_HASH	SYNC	
SIMPLE_EXPRESSION	TERM SIMPLE_EXPRESSION_DASH	SYNC	
TERM	FACTOR TERM_DASH	SYNC	
FACTOR	id	SYNC	
SIGN	SYNC		
STATEMENT_LIST_DASH	STATEMENT STATEMENT_LIST_DASH		STATEMENT STATEMENT_LI
SIMPLE_EXPRESSION_DASH		\L	
TERM_DASH		\L	
EXPRESSION_HASH		\L	

Parser output:



Parsing Result

Auto-Saved to: output/parsing.txt

Stack
$ METHOD_BODY
$ STATEMENT_LIST
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH DECLARATION
$ STATEMENT_LIST_DASH ; id PRIMITIVE_TYPE
$ STATEMENT_LIST_DASH ; id int
$ STATEMENT_LIST_DASH ; id
$ STATEMENT_LIST_DASH ;
$ STATEMENT_LIST_DASH
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH ASSIGNMENT
$ STATEMENT_LIST_DASH ; EXPRESSION assign id
$ STATEMENT_LIST_DASH ; EXPRESSION assign
$ STATEMENT_LIST_DASH ; EXPRESSION
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH FACTOR
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH num
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH SIMPLE_EXPRESSION_DASH
$ STATEMENT_LIST_DASH ; EXPRESSION_HASH
$ STATEMENT_LIST_DASH ;
$ STATEMENT_LIST_DASH
$ STATEMENT_LIST_DASH STATEMENT
$ STATEMENT_LIST_DASH IF
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION (if
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION (
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION_HASH SIMPLE_EXPRESSION
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM
$ STATEMENT_LIST_DASH } STATEMENT { else } STATEMENT {) EXPRESSION_HASH SIMPLE_EXPRESSION_DASH TERM_DASH FACTOR