2017-2018

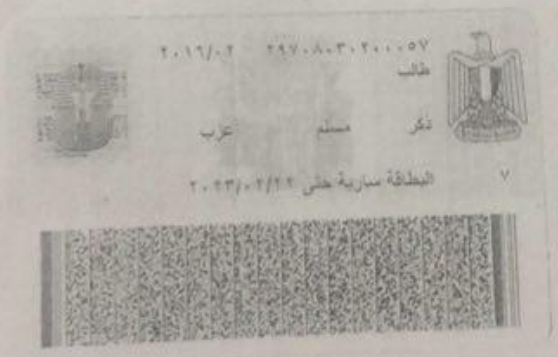
كلية الهندسة

الاسم : يوسف محمد فتحى حسن على

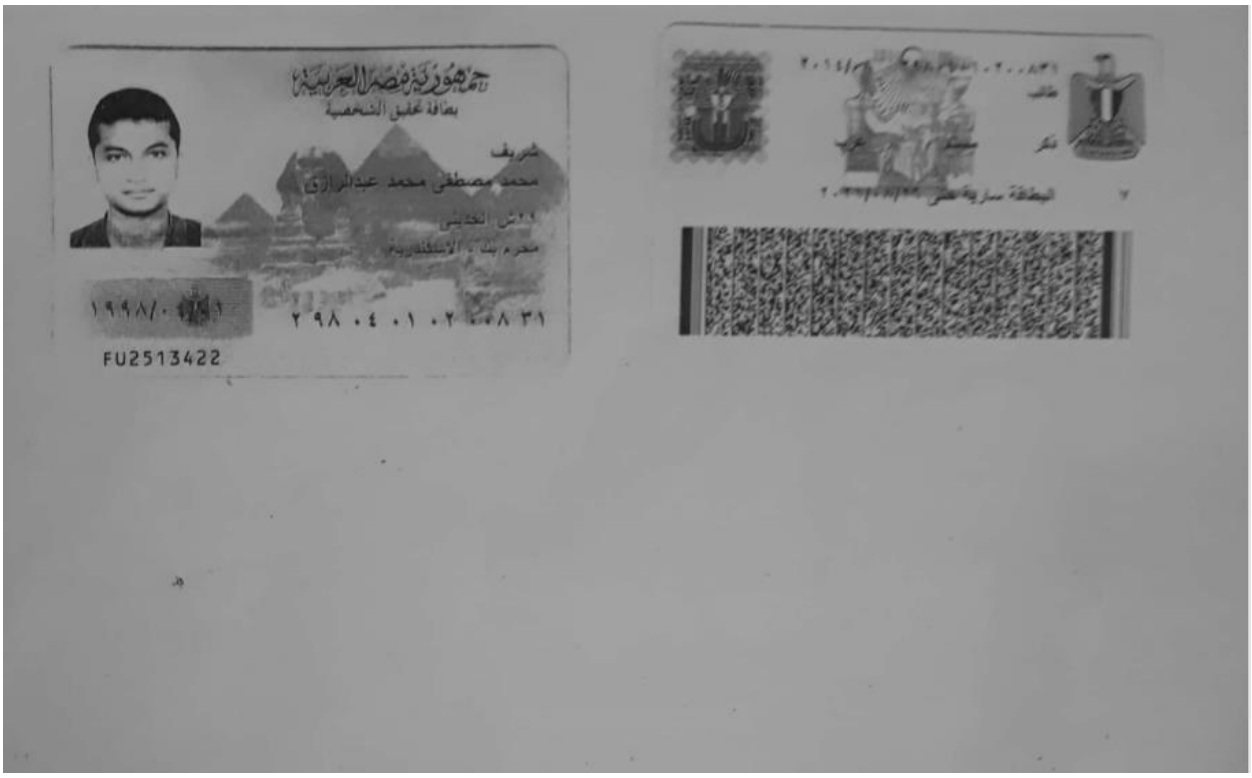
القسم : هندسة الحاسب والنظم

الرقم القومي : 29708030200057


جمهورية مصر العربية
بطاقة تحقيق الشخصية
يوسف
محمد فتحى حسن على
١٨ش ال ياسر خ١٣ابراهيم العطار
اول الرمل - الاسكندرية
٢٩٧٠٨٠٣٠٢٠٠٠٥٧
GJ4927384


٢٩٧٠٨٠٣٠٢٠٠٠٥٧
٢٠١٦/٠٢
طالب
ذكر
مسلم
عرب
٧
البطاقة سارية حتى ٢٠٢٣/٠٢/٢٢

Name: Sherif Mohamed Mostafa **ID: 22**



جمهورية مصر العربية

بطاقة تحقيق الشخصية

احمد

على عبدالمجيد على حسنين

٥ ش ههيا-كامب شيزار

باب شرق - الاسكندرية

٢٩٧١٠٠٦٠٢٠٠٣٥٨

FM5598352

٢٩٧١٠٠٦٠٢٠٠٣٥٨ ٢٠١٣/١١

طالب

ذكر مسلم اعزب

٧ البطاقة سارية حتى ٢٠٢٠/١١/٢٦

2017-2018

كلية الهندسة

الاسم : احمد على عبد المجيد على حسنين

القسم : هندسة الحاسب والنظم

الرقم القومي : 29710060200358

كلية الهندسة

جامعة الإسكندرية

FACULTY OF ENGINEERING

ALEXANDRIA UNIVERSITY

جمهورية مصر العربية

بطافة تحقيق الشخصية

حازم

مرسى حسن مرسى صالح

١٣ش عبداللطيف النشار-الاقبال

اول الرمل - الاسكندرية

١٩٩٧/٠٩٢١

٢٩٧٠٩٢١٠٢٠٢٠٩٧

HI2863442

٢٩٧٠٩٢١٠٢٠٢٠٩٧ ٢٠١٨/٠١

طالب

ذكر مسلم أعزب

٧ البطاقة سارية حتى ٢٠٢٥/٠١/٢٣

Scanned by TapScanner

2017-2018

كلية الهندسة

الاسم : حازم مرسى حسن مرسى صالح

القسم : هندسة الحاسب والنظم

الرقم القومي : 29709210202097

كلية الهندسة
جامعة الاسكندرية
FACULTY OF ENGINEERING
ALEXANDRIA UNIVERSITY

Scanned by TapScanner

Compilers

Final project

Compiler front end

Team members

Name: Ahmed Ali Abd El Meguid Ali Hassanine ID: 7

Name: Hazem Morsy Hassan Morsy ID: 18

Name: Sherif Mohamed Mostafa Mohamed ID: 22

Name: Youssef Mohamed Fathy Hassan Aly ID: 66

Abstract:

Our project provides tools for aiding compiler and language specification design. The input is lexical rules, syntax grammar (CFG) and a program snippet. Output is data concerning lexical analysis and syntax analysis. An approach was used to integrate the semantic analyzer together with an intermediate code generator to generate intermediate code specific to the JAVA language. The generated code is in the JAVA Byte code format and covers simple JAVA code fragments only as variable declaration and assignment. The algorithms followed in the program are according to the formal language specification and analysis, using techniques as deterministic finite automaton for pattern matching and parser table for syntax analysis. The program offers a convenient method for testing language specifications and also for determining the lexical and syntax analysis output for an input following these specifications. First, regular expressions definition of the language is processed into a finite automaton structure to match the input with tokens and determine any tokenizing errors. Then the syntax grammar specification of the language is processed into a parsing table that can be used together with the tokens stream to obtain the leftmost derivation of the input. Then, according to explicitly hardcoded JAVA semantic rules, some of the supported JAVA language code can be input to be converted to JAVA Byte code.

Objective:

The project aims to building a front end compiler generator code.The program can be subdivided into three phases with the following objectives:

- phase I aims to build an automatic lexical analyzer generator tool.
- phase II aims to build an automatic parser generator tool.
- phase III aims to test the program in generating JAVA Byte code for a subset of the JAVA language.

Introduction:

It goes without a doubt that constructing a unified structured way of understanding between the programmer and the computer is of major importance in the computer science field. A compiler is generally computer software that translates a high level programming language to a more lower level programming language (may be even machine language) that can be better understood by the computer. Designing a compiler is generally based on having a formal specification for the programming language and its associated grammar and semantics. In this project we have designed a “Compiler generator tool” which given the lexical rules, syntax rules (context free grammar) and semantics rules of the programming language specification, the corresponding byte code (or assembly code) is generated together with some error handling for lexical (tokens) or syntax errors. This project implements the first four phases of the compiler generating a lexical analyzer based on deterministic finite automaton, a syntax analyzer based on a non recursive predictive parser, a semantic analyzer and a test case generation for intermediate code of JAVA (JAVA Byte Code). The report covers some more implementation details and provides sample runs for our program. For further details about compilers and languages refer to the references.

Literature review:

Compiler generator tools have been used for some time and surely their importance and impact is significant in designing and testing programming languages and their specifications. Even the earlier phases of a compiler can be used in operations such as string parsing and pattern matching using finite automata. The tools for compiler generation can be classified into:

- Scanner generators: given regular expressions description for the language, it generates a finite automaton to recognize these regular expressions and identify the tokens in the input. It also provides any errors in tokenizing the input. A well known example for scanner generators is **LEX** made by *M. E. Lesk and E. Schmidt*, which we provided a report for how to use it and we found to be of a high level of practicality.
- Parser Generator: produces syntax analyzers (as a non predictive parser) given grammatical description of the language. Useful to convert a stream of tokens obtained from the lexical analyzer to a set of productions deriving the input. Example of parser generator: **BISON** by *Charles Donnelly and Richard Stallman.*
- Semantic analyzer: analyzes the given semantic rules to define the actions performed to create an annotated parse tree that can be used in generation of intermediate code.
- Syntax directed translation engines: Generates intermediate code with the three address format taking as input a parse tree.
- Code optimizer: performs optimization on the generated intermediate code such as loop unrolling and rescheduling the code to provide pipelining capabilities.
- Code generator: this part generates the target machine language given the optimized intermediate code and the machine specifications.

What we provide in this project is a mixture of some of these tools. Our project contains a scanner generator tool, a parser generator tool and an integrated semantic analyzer - syntax directed translation engine tool specific for generation of JAVA Byte code for some java code snippets. Further work can be done on the project to support all of the JAVA byte code and also integrate support for more languages.

A note is the simplicity of using our program, corresponding to other tools which are more complex to use. However, our program is suited for simpler use which is mainly concerning the aid of language specifications design, pattern matching and help in compilation of some simple JAVA codes. More details about our projects usability and features will be provided throughout the report.

Problem statement:

In this project it is required to implement the code for generating a compiler front end. Moreover, for a chosen subset of the java language it is required to produce the intermediate code of the given input in the form of JAVA byte-code.

For (phase I) we need to implement a lexical analyzer generator tool which constructs NFAs for the given regular expressions then combining them together and converting it to minimized DFA and we need to print the minimized DFA transition table. The generated lexical analyzer has to read the input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions and we need to put each identifier in the symbol table.

For (phase II) we need to implement an (LL1) parser generator tool and then compute the first and follow sets for each terminal and we use these sets to build the parser table which helps in connecting each terminal to a non terminal and check if it contains error or a specific production. if an error is encountered we need to handle it using the panic mode recovery.

For (phase III) we need to provide support for the generation of JAVA Byte code by making proper semantic analysis and intermediate code generation actions.

Phase I

Data structures:

Parser:

Vector of string (input): used to carry rules to be parsed each in separated entries.

Vector of char (punctuations): used to carry punctuation symbols.

Vector of char (reservedSymbols): used to carry reserved symbols.

Struct (regularDefinition): used to hold identifier name and postfix nodes of the definition.

Struct (regularExpression): used to hold identifier name and postfix nodes of the expression.

Struct (keyWords): used to hold keyword name and its postfix.

Class (Node) : used to hold data about each element in postfix expression (value of operand or operation, Boolean to check if it is unary operation and precedence of the operation).

Grammar:

String Identifier: this string holds the corresponding identifier token and is used to fill in the symbol table.(global variable).

Vector of strings (symbol_table): used when performing the lexical analysis of the given input to identify the token values corresponding to identifier and keep them in the symbol table.(global variable).

Class(Grammar): This class, (a singleton class), holds the data structures and functions to manipulate the given expressions and keep their data as:

- Vector for the keywords.
- Vector for the punctuation symbols.
- Vector for the regular definitions.
- Vector for the reserved symbols.
- Vector of NFAs corresponding to the productions of regular expressions, keywords, punctuation symbols and any sentences formed from this grammar's alphabet.
- Some setters and getters for the data structures and utility functions to add or remove elements.

The state machines:

Struct(node): this is the structure identifying the node used in the graphs of the state machines it contains:

- An integer indicating it's number and it is unique for each node and kept as an invariant that in any machine each node has a unique number identifying it.
- A map of transitions with its key as an input character and its value is a vector of nodes (assuming the general nfa case where a single input can go to several nodes).
- A token string showing whether or not this node is an acceptance state and if so it keeps the corresponding token to this final state.

Class(NFA): Although its name is a bit misleading, this class is used in implementing both the nfa and dfa alike with the only difference is the implicit fact that each node in a dfa has a single transition (if exists) for an input. Represented here by the fact that all the node vectors in the transitions maps of the nodes have a single entry. This class has the main data structures and functions needed in implementing the state machine:

- A pointer to a node representing the current state this machine is at, set to the start node at machine creation.
- A pointer to the start node (a single start state according to the needed implementation).
- The number of states in this machine.
- A vector of node pointers for the final state nodes.
- A vector of node pointers for the non-final state nodes.
- A vector for all states in this machine (all nodes).
- Setters and getters and utility functions to add elements,change the machine state given an input (used in DFAs), get the token if the current state is final, push the states of another machine to this machine (used in combining the nfa machines) and a function that removes any repetitions in the node vectors used as a utility function in some algorithms.

DFA:

map<int,vector<node*> > equivalent : contains all dfa nodes and their corresponding nfa nodes

vector<node*> current_equivalent : vector contains all nfa nodes that corresponds to a dfa node, keeps changing while looping through the nfa nodes to create the dfa nodes separately

Functions and algorithms:

Parser:

The main idea of parsing input is the conversion of rules from infix expressions to postfix expressions.

We construct a class Node to represent each element in postfix expression which has: Name, precedence (used for operation).

The method getNode takes an expression and converts it to a vector of nodes in the postfix form. Each node carries a character or an operation which is the base idea of NFA creation.

The target behind the conversion of infix to postfix expression is to achieve the precedence of “AND” operation over “OR” Operation between Operands.

All possible inputs for all types of rules are taken under consideration to deal with each type separately.

The parsing process begins with reading rules from file and parsing it line by line.

The following methods are used for this process:

read_file (string fileName): it reads rules from a text file and returns them in vector.

parseInput(): Regex is used for capturing groups of all types of rules.

First, if the rule is punctuation, then we save each symbol in the punctuation vector.
Second, if the rule is a keyword, we create w postfix expression nodes for each word by ANDing all its characters and store it with keyword name atomically in a struct then store this struct in a vector of keyWords.
Third, if the rule was a regular expression or a regular definition, the identifier is saved in struct along with its postfix nodes. We use the following method to parse the operands and return its postfix form.

getNodes(string line): Used to parse the line for operands expression.
The design of the getNode method depends on the Shunting-yard algorithm. It is a stack-based algorithm for the conversion of infix to postfix form.

Regex is used here to get the first operand, operation and second operand if any.

We loop on the expression and separate operands until the last one.
We have three possible values for operand: bracket, ranges and any operand other than the previous ones.
Brackets are recursively parsed, ranges are stored as all range values ORed together in a postfix expression. Any other operand is maybe a new one or a reserved before. If it is new, we separate each character and ANDing them and make a recursive call with this expression, otherwise, we simply reference back to the pre-identified expression or identifier in vectors containing them and get their postfix expression.

We also check for any reserved symbol (any character preceded with a backslash) while parsing operands and punctuations

get_betweenbrackets(string str): it takes a string (str) as a parameter that contains opening bracket at its first character then we use stack to match the opening bracket with its closing one and check for the next character after the closing bracket if it contains a unary operation (+ , *).

Now , we constructed the first operand to be passed again to the recursive function (getNodes) after removing the brackets ,then we continue parsing this string to get the next binary operation (and , or) and the second operand which are used (if exist) also by the function (getNodes).

We use this method separately from regex as regex failed to capture nested brackets. Hence, this handles nested brackets perfectly using a stack-based algorithm.

NFA creation:

Here we take a look at the main functions used in building the combined NFA from the parsed input obtained in the parsing stages, note that the said invariant about the node numbers in a machine is kept:

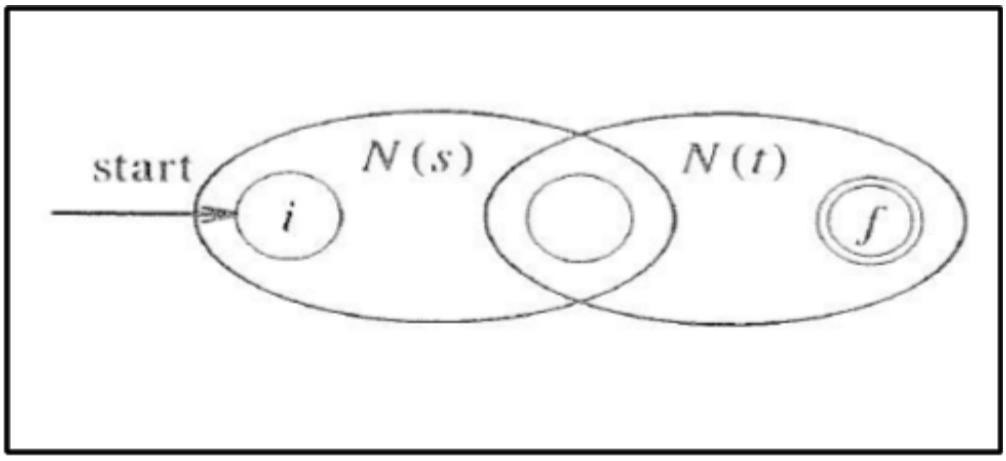
- **Create_machine:** this function takes in an input character and token, it then creates two nodes one as the start node with no token and one as the final with the input token, it joins those two nodes with a transition on the given input and returns a machine corresponding to the created nodes. It is the first stage in the creation of the nfa where every input symbol from the alphabet has a corresponding machine created. According to the expression this symbol is in, its token is assigned.
- **Star_op:** this function takes a machine and performs the star operation on it using the same technique studied in lecture. It adds a new start node and new final node. It removes the token from the original machine making its final states regular states and adds this token to the new final state. It adds the corresponding empty string transitions: from former final states to new final state, from new start state to new final state and from former final states to the former start state.

- **Conc_op:** given two state machines it returns a new machine corresponding to their concatenation. Done by simply adding empty string transitions from the back machine's final states to the front machine's start state. It then removes the tokens from the back machine's final states and sets the final states of the resulting machine as those of the front machine only.
- **Or_op:** this function returns the combination of two state machines using the combine function that is to be shown later.
- **Plus_op:** this function performs the plus operation given a machine. It simply concatenates a copy of this machine to the machine resulting from performing the star operation on it and returns the result.
- **Combine:** this function takes in a vector of machines and returns their combination as a single machine. It simply creates a new start node for the resultant machine then adds transitions from

this start node to the start nodes of the machines to be combined on empty string, then fills in the resulting machine data and returns it.

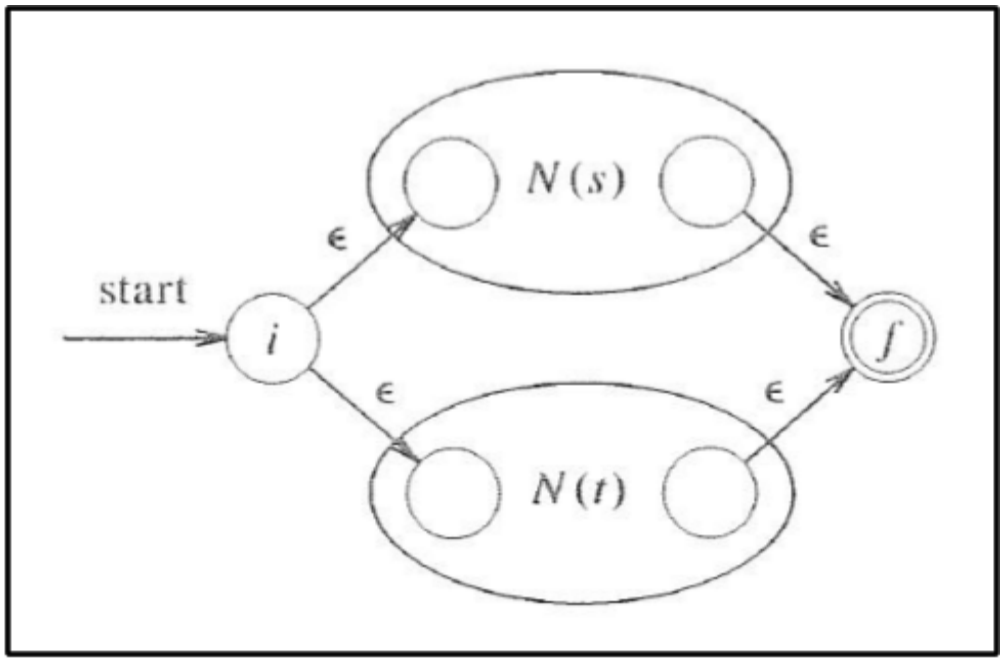
Here a question is unanswered, how do we take care of the priority according to the order of defining the expressions? Another invariant was taken into consideration, when combining machines, the less

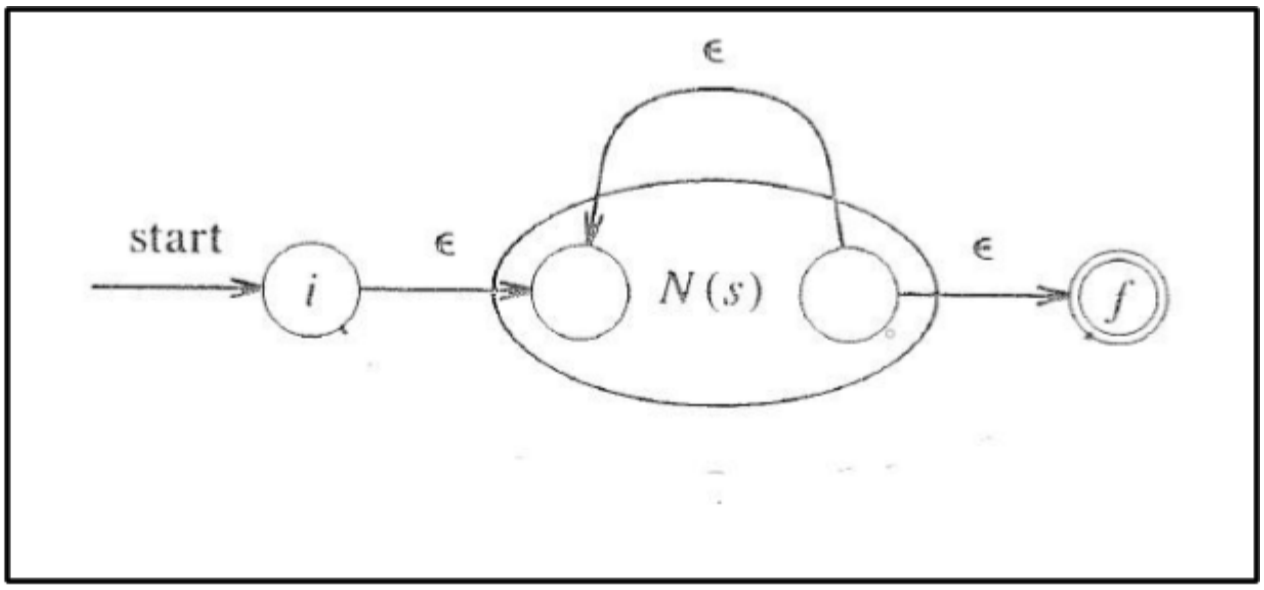
node number gives it higher priority that means that this machine was defined by a regular expression older than other machines having nodes with larger node numbers.



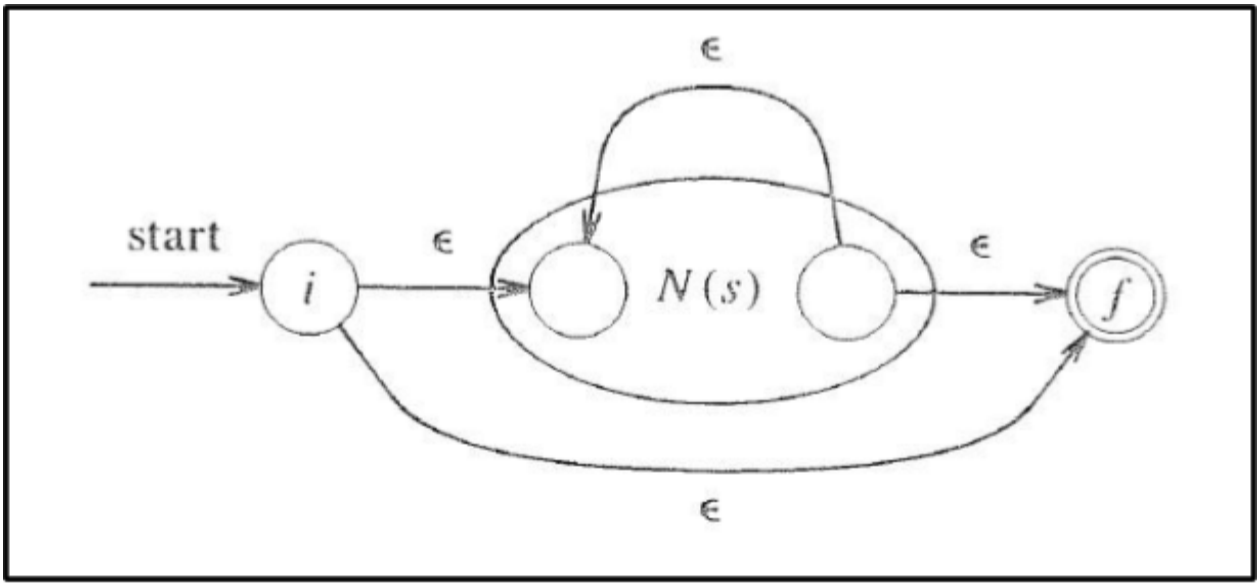
conc_op

or_op





Plus_op



star_op

Get_nfa_expression (string token,vector < Node > postfix) :

It is the main function which forms the NFA machine according to the nodes of expression tree which is represented in postfix form.it depends on the four main machines of basic operation then, merge them together (a|b , a b , a* , a+) keeping the precedence.

It is based on a stack to process the postfix expression. We check if the current node contains an operand then we create a machine for this and push it to stack with the name of its token else , then we pop a machine or two from the stack according to type of operation (unary or binary respectively) to perform the operation for these machine(s) and push the result machine again to the stack. We repeat these steps until the stack is empty (postfix expression ended).

Get_NFA_all_rules(): the main goal in this function is to get NFAs of all used rules of the input file (keywords , punctuations , expressions) in this mentioned order

to give higher priorities to keywords and punctuations and form one machine only to pass the input to this machine after minimization.

We create NFA machines for (keywords , punctuations , expressions) , vector for all keywords , one for all expressions and one for punctuation symbols merging them in one vector of these NFAs to be combined and used as one NFA machine.

DFA creation:

Here we take a look at the main function performing the conversion of nfa to dfa, taking advantage of the defined invariants to obtain the correct final states and transitions:

NFA build_dfa(NFA nfa) : the main function that creates the dfa diagram using the nfa transition table which can be accessed through the nfa nodes. It works as follows:

- **Initialization stage:** here the dfa is initialized as follows, a start node is created by obtaining the equivalent dfa node corresponding to the equivalent nodes to the start node of the input nfa and set to be the start of this dfa. Note that for each created node in the dfa a vector of its corresponding nfa nodes is kept to be used throughout this algorithm.
- **The main stage:** in this stage the main conversion takes place. For each added dfa node we obtain an equivalent dfa node for each of its transitions. These transitions are the transitions of the corresponding nfa nodes equivalent to this dfa node. Note that the returned nodes for each transition are nfa nodes as these are the transitions in the equivalent nfa nodes, so after finishing each transitions, we obtain the equivalent dfa node for all these nodes by first obtaining the equivalent nfa nodes for each of the resulting nodes then obtaining an equivalent dfa node using the functions described later.
- **The ending stage:** having finished all the dfa nodes and all the transitions and set all the needed data in this dfa, we set the number of states in this dfa according to the number of dfa nodes needed to be created and return this dfa machine.

string get_token_least(vector<node*>nodes) : returns the token of the node with the lowest number corresponding to the highest priority to set it to the dfa node.

node * get_equivalent_dfa_node(NFA dfa,vector<node *>nfa_nodes) : returns the dfa node corresponding to this set of nfa nodes if it already exists in this DFA machine or a new node with a unique number.

int equivalent_number(vector<node*> nfa_nodes) : function to generate a unique number for the new dfa node to be created based on the numbers of the corresponding nfa nodes and the size of the vector, repetitions excluded. It uses a well known method called the Cantor pairing function which is $F(A,B) = ½*(A+B)*(A+B+1) + A$ that generates a unique integer given an integer pair. We use this function in assigning a unique DFA node number to represent a set of NFA nodes by performing iteratively the Cantor function and obtaining a unique number corresponding to these nfa nodes' numbers.

vector<node*> get_equivalent_nodes(node* n,vector<node*>result) : a **DFS** algorithm to get the output nfa states of the state 'n' on epsilon (empty input). Used to get the equivalent nfa nodes that should be represented as a single DFA node.

vector<node*> remove_repetitions(vector<node*>nodes) : function to remove repeated nodes in the given vector. Used in building the DFA to decrease the running time of the algorithm as we don't need to find equivalent nodes of a repeated node.

DFA minimization:

Here we show how after obtaining the DFA we minimize its states. This algorithm mainly performs the partitioning algorithm but with the following difference: in the first step, the final states partition is further partitioned according to the corresponding token where not all final states correspond to the same token.

minimize(NFA machine): given a dfa machine, it returns a minimized machine for it by performing the following steps:

- **Initialization:** in this part, the initial partitions are obtained by partitioning the final states according to their tokens and keeping track of each partition. Then a partition is added for the non final states if any exists. Note that throughout this algorithm, any node that is to be placed in a partition, its number is changed to be equal to this partition's number so in the final dfa resulting, its nodes have numbers representing the partition they were in at the end of the algorithm.
- **Main part:** in this part the heart of the minimization algorithm is performed. For each partition, each node in this partition is checked whether or not it needs to change its partition, this is done by checking that this node is equivalent to all the nodes in this partition, if not then it needs to change its partition. Then if this node needs to be partitioned, first we check for each new partition in this iteration whether or not it is the partition for this node, this is checked by seeing whether or not the node is equivalent to a node in this partition. If no such partition is found, a new partition is created for this node. We kept doing so till no new partition was added.
- **The final part:** We create a minimized machine and fill in its nodes where each node represents a partition (one node from each partition), we then fill the data of this machine step by step (its final,non final states and initial state etc) we then set the number of nodes of this machine according to the number of partitions and return it.

Equivalent: this function takes two nodes and checks whether or not they are equivalent, used throughout the minimization algorithm. It checks that the two nodes have the same token value corresponding and go to the same nodes on each transition, if not, then they are not equivalent.

Final parser:

get_next_token(string input, NFA dfa):

the function loops through the input string to classify each token in it. It takes each character of string and passes it to the minimized DFA machine to apply the transition until it reaches a final state or escape character ('\n' , '\t' , ' ') or it has no more transitions.

Now , the token may be an error (if it ends in a non-final state) or the token of the last final state reached; to apply the maximal munch.

It returns a pair of strings which holds the token and its value.

print_output(string input, NFA dfa):

It is used for printing the lexical analyzer output in the output file by looping through the input string until it becomes empty and calling the get_token function to split the string according to the result of this function and reparse the string after the value of last token.

Minimal DFA transition table (for rules in the problem statement):

initial state: 43

from 43 with input ! to 44

from 43 with input (to 0

from 43 with input) to 1

from 43 with input * or / to 2

from 43 with input + or - to 3

from 43 with input , to 4

from 43 with input digit to 19

from 43 with input ; to 6

from 43 with input < to 22

from 43 with input = to 8

from 43 with input > to 23

from 43 with input [A-Z] or [a-z] - {b,e,f,i,w} to 24

from 43 with input b to 25

from 43 with input e to 26

from 43 with input f to 27

from 43 with input i to 28

from 43 with input w to 29

from 43 with input { to 10

from 43 with input } to 11

from 44 with input = to 7

from 19 with input . to 45

from 19 with input digit to 19

from 22 with input = to 7

from 8 with input = to 7

from 23 with input = to 7

from 24 with input digit to 24

from 24 with input [A-Z] or [a-z] to 24

from 25 with input digit to 24

from 25 with input [A-Z] or [a-z] - {o} to 24

from 25 with input o to 30

from 26 with input digit to 24

from 26 with input [A-Z] or [a-z] - {l} to 24

from 26 with input l to 31

from 27 with input digit to 24

from 27 with input [A-Z] or [a-z] - {l} to 24

from 27 with input l to 32

from 28 with input digit to 24

from 28 with input [A-Z] or [a-z] - {f,n} to 24

from 28 with input f to 12

from 28 with input n to 33

from 29 with input digit to 24

from 29 with input [A-Z] or [a-z] - {h} to 24

from 29 with input h to 34

from 45 with input digit to 46

from 30 with input digit to 24

from 30 with input [A-Z] or [a-z] - {o} to 24

from 30 with input o to 35

from 31 with input digit to 24

from 31 with input [A-Z] or [a-z] - {s} to 24

from 31 with input s to 36

from 32 with input digit to 24

from 32 with input [A-Z] or [a-z] - {o} to 24

from 32 with input o to 37

from 12 with input digit to 24

from 12 with input [A-Z] or [a-z] to 24

from 33 with input digit to 24

from 33 with input [A-Z] or [a-z] - {t} to 24

from 33 with input t to 13

from 34 with input digit to 24

from 34 with input [A-Z] or [a-z] - {i} to 24

from 34 with input i to 38

from 46 with input digit to 56

from 46 with input E to 66

from 46 with input \ to 18

from 47 with input 0 to 56

from 47 with input 1 to 57

from 47 with input 2 to 58

from 47 with input 3 to 59

from 47 with input 4 to 60

from 47 with input 5 to 61

from 47 with input 6 to 62

from 47 with input 7 to 63

from 47 with input 8 to 64

from 47 with input 9 to 65

from 47 with input E to 66

from 47 with input \ to 18

from 48 with input 0 to 56

from 48 with input 1 to 57

from 48 with input 2 to 58

from 48 with input 3 to 59

from 48 with input 4 to 60

from 48 with input 5 to 61

from 48 with input 6 to 62

from 48 with input 7 to 63

from 48 with input 8 to 64

from 48 with input 9 to 65

from 48 with input E to 66

from 48 with input \ to 18

from 49 with input 0 to 56

from 49 with input 1 to 57

from 49 with input 2 to 58

from 49 with input 3 to 59

from 49 with input 4 to 60

from 49 with input 5 to 61

from 49 with input 6 to 62

from 49 with input 7 to 63

from 49 with input 8 to 64

from 49 with input 9 to 65

from 49 with input E to 66

from 49 with input \ to 18

from 50 with input 0 to 56

from 50 with input 1 to 57

from 50 with input 2 to 58

from 50 with input 3 to 59

from 50 with input 4 to 60

from 50 with input 5 to 61

from 50 with input 6 to 62

from 50 with input 7 to 63

from 50 with input 8 to 64

from 50 with input 9 to 65

from 50 with input E to 66

from 50 with input \ to 18

from 51 with input 0 to 56

from 51 with input 1 to 57

from 51 with input 2 to 58

from 51 with input 3 to 59

from 51 with input 4 to 60

from 51 with input 5 to 61

from 51 with input 6 to 62

from 51 with input 7 to 63

from 51 with input 8 to 64

from 51 with input 9 to 65

from 51 with input E to 66

from 51 with input \ to 18

from 52 with input 0 to 56

from 52 with input 1 to 57

from 52 with input 2 to 58

from 52 with input 3 to 59

from 52 with input 4 to 60

from 52 with input 5 to 61

from 52 with input 6 to 62

from 52 with input 7 to 63

from 52 with input 8 to 64

from 52 with input 9 to 65

from 52 with input E to 66

from 52 with input \ to 18

from 53 with input 0 to 56

from 53 with input 1 to 57

from 53 with input 2 to 58

from 53 with input 3 to 59

from 53 with input 4 to 60

from 53 with input 5 to 61

from 53 with input 6 to 62

from 53 with input 7 to 63

from 53 with input 8 to 64

from 53 with input 9 to 65

from 53 with input E to 66

from 53 with input \ to 18

from 54 with input 0 to 56

from 54 with input 1 to 57

from 54 with input 2 to 58

from 54 with input 3 to 59

from 54 with input 4 to 60

from 54 with input 5 to 61

from 54 with input 6 to 62

from 54 with input 7 to 63

from 54 with input 8 to 64

from 54 with input 9 to 65

from 54 with input E to 66

from 54 with input \ to 18

from 55 with input 0 to 56

from 55 with input 1 to 57

from 55 with input 2 to 58

from 55 with input 3 to 59

from 55 with input 4 to 60

from 55 with input 5 to 61

from 55 with input 6 to 62

from 55 with input 7 to 63

from 55 with input 8 to 64

from 55 with input 9 to 65

from 55 with input E to 66

from 55 with input \ to 18

from 35 with input digit to 24

from 35 with input [A-Z] or [a-z] - {l} to 24

from 35 with input l to 39

from 36 with input digit to 24

from 36 with input [A-Z] or [a-z] - {e} to 24

from 36 with input e to 14

from 37 with input digit to 24

from 37 with input [A-Z] or [a-z] - {a} to 24

from 37 with input a to 40

from 13 with input digit to 24

from 13 with input [A-Z] or [a-z] to 24

from 38 with input digit to 24

from 38 with input [A-Z] or [a-z] - {l} to 24

from 38 with input l to 41

from 56 with input 0 to 56

from 56 with input 1 to 57

from 56 with input 2 to 58

from 56 with input 3 to 59

from 56 with input 4 to 60

from 56 with input 5 to 61

from 56 with input 6 to 62

from 56 with input 7 to 63

from 56 with input 8 to 64

from 56 with input 9 to 65

from 56 with input E to 66

from 56 with input \ to 18

from 57 with input 0 to 56

from 57 with input 1 to 57

from 57 with input 2 to 58

from 57 with input 3 to 59

from 57 with input 4 to 60

from 57 with input 5 to 61

from 57 with input 6 to 62

from 57 with input 7 to 63

from 57 with input 8 to 64

from 57 with input 9 to 65

from 57 with input E to 66

from 57 with input \ to 18

from 58 with input 0 to 56

from 58 with input 1 to 57

from 58 with input 2 to 58

from 58 with input 3 to 59

from 58 with input 4 to 60

from 58 with input 5 to 61

from 58 with input 6 to 62

from 58 with input 7 to 63

from 58 with input 8 to 64

from 58 with input 9 to 65

from 58 with input E to 66

from 58 with input \ to 18

from 59 with input 0 to 56

from 59 with input 1 to 57

from 59 with input 2 to 58

from 59 with input 3 to 59

from 59 with input 4 to 60

from 59 with input 5 to 61

from 59 with input 6 to 62

from 59 with input 7 to 63

from 59 with input 8 to 64

from 59 with input 9 to 65

from 59 with input E to 66

from 59 with input \ to 18

from 60 with input 0 to 56

from 60 with input 1 to 57

from 60 with input 2 to 58

from 60 with input 3 to 59

from 60 with input 4 to 60

from 60 with input 5 to 61

from 60 with input 6 to 62

from 60 with input 7 to 63

from 60 with input 8 to 64

from 60 with input 9 to 65

from 60 with input E to 66

from 60 with input \ to 18

from 61 with input 0 to 56

from 61 with input 1 to 57

from 61 with input 2 to 58

from 61 with input 3 to 59

from 61 with input 4 to 60

from 61 with input 5 to 61

from 61 with input 6 to 62

from 61 with input 7 to 63

from 61 with input 8 to 64

from 61 with input 9 to 65

from 61 with input E to 66

from 61 with input \ to 18

from 62 with input 0 to 56

from 62 with input 1 to 57

from 62 with input 2 to 58

from 62 with input 3 to 59

from 62 with input 4 to 60

from 62 with input 5 to 61

from 62 with input 6 to 62

from 62 with input 7 to 63

from 62 with input 8 to 64

from 62 with input 9 to 65

from 62 with input E to 66

from 62 with input \ to 18

from 63 with input 0 to 56

from 63 with input 1 to 57

from 63 with input 2 to 58

from 63 with input 3 to 59

from 63 with input 4 to 60

from 63 with input 5 to 61

from 63 with input 6 to 62

from 63 with input 7 to 63

from 63 with input 8 to 64

from 63 with input 9 to 65

from 63 with input E to 66

from 63 with input \ to 18

from 64 with input 0 to 56

from 64 with input 1 to 57

from 64 with input 2 to 58

from 64 with input 3 to 59

from 64 with input 4 to 60

from 64 with input 5 to 61

from 64 with input 6 to 62

from 64 with input 7 to 63

from 64 with input 8 to 64

from 64 with input 9 to 65

from 64 with input E to 66

from 64 with input \ to 18

from 65 with input 0 to 56

from 65 with input 1 to 57

from 65 with input 2 to 58

from 65 with input 3 to 59

from 65 with input 4 to 60

from 65 with input 5 to 61

from 65 with input 6 to 62

from 65 with input 7 to 63

from 65 with input 8 to 64

from 65 with input 9 to 65

from 65 with input E to 66

from 65 with input \ to 18

from 66 with input 0 to 20

from 66 with input 1 to 20

from 66 with input 2 to 20

from 66 with input 3 to 20

from 66 with input 4 to 20

from 66 with input 5 to 20

from 66 with input 6 to 20

from 66 with input 7 to 20

from 66 with input 8 to 20

from 66 with input 9 to 20

from 18 with input L to 21

from 39 with input digit to 24

from 39 with input [A-Z] or [a-z] - {e} to 24

from 39 with input e to 42

from 14 with input digit to 24

from 14 with input [A-Z] or [a-z] to 24

from 40 with input digit to 24

from 40 with input [A-Z] or [a-z] - {t} to 24

from 40 with input t to 15

from 41 with input digit to 24

from 41 with input [A-Z] or [a-z] - {e} to 24

from 41 with input e to 16

from 20 with input digit to 5

from 42 with input digit to 24

from 42 with input [A-Z] or [a-z] - {a} to 24

from 42 with input a to 9

from 15 with input digit to 24

from 15 with input [A-Z] or [a-z] to 24

from 16 with input digit to 24

from 16 with input [A-Z] or [a-z] to 24

from 5 with input digit to 5

from 9 with input digit to 24

from 9 with input [A-Z] or [a-z] - {n} to 24

from 9 with input n to 17

from 17 with input digit to 24

from 17 with input [A-Z] or [a-z] to 24

final states:

0 result: (

1 result:)

2 result: mulop

3 result: addop

4 result: ,

19 result: num

6 result: ;

22 result: relop

8 result: assign

23 result: relop

24 result: id

25 result: id

26 result: id

27 result: id

28 result: id

29 result: id

10 result: {

11 result: }

7 result: relop

30 result: id

31 result: id

32 result: id

12 result: if

33 result: id

34 result: id

35 result: id

36 result: id

37 result: id

13 result: int

38 result: id

39 result: id

14 result: else

40 result: id

41 result: id

20 result: num

21 result: num

42 result: id

15 result: float

16 result: while

5 result: num

9 result: id

17 result: boolean

number of states: 67

Sample Runs:

For the input file in the pdf:

```
Lexical Output:
token: int value: int
token: id value: sum
token: , value: ,
token: id value: count
token: , value: ,
token: id value: pass
token: , value: ,
token: id value: nmt
token: ; value: ;
token: while value: while
token: ( value: (
token: id value: pass
token: relop value: !=
token: num value: 10
token: ) value: )
token: { value: {
token: id value: pass
token: assign value: =
token: id value: pass
token: addop value: +
token: num value: 1
token: ; value: ;
token: } value: }

Symbols in symbol table:
1 : pass
2 : nmt
3 : count
4 : sum
```

Test 1:

rules.txt - Notepad

File Edit Format View Help

```
[; , \( \) { }]
small = a - z
capital = A - Z
letter = small | capital
digit = 0 - 9
digits : digit+
identifier : letter+ | letter digit*
{boolean int float if else while for}
math_op : \+ | - | \* + / | =
relop: \=\= | !\= | > | >\= | < | <\=
```

input.txt - Notepad

File Edit Format View Help

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

output.txt - Notepad

File Edit Format View Help

```
Lexical Output:
token: int value: int
token: identifier value: sum
token: , value: ,
token: identifier value: count
token: , value: ,
token: identifier value: pass
token: , value: ,
token: identifier value: mnt
token: ; value: ;
token: while value: while
token: ( value: (
token: identifier value: pass
token: relop value: !=
token: digits value: 10
token: ) value: )
token: { value: {
token: identifier value: pass
token: mathop value: =
token: identifier value: pass
token: mathop value: +
token: digits value: 1
token: ; value: ;
token: } value: }
```

Test 2:

rules.txt - Notepad

File Edit Format View Help

```
[; , \( \) s]
small = a - z
capital = A - Z
letter = small | capital
identifier : letter+ | letter digit*
{boolean int}
```

input.txt - Notepad

File Edit Format View Help

```
int sum , count , pass ,
mnt; while (pass)
{
}
```


output.txt - Notepad
File Edit Format View Help
Lexical Output:
token: int value: int
token: identifier value: sum
token: , value: ,
token: identifier value: count
token: , value: ,
token: identifier value: pass
token: , value: ,
token: identifier value: mnt
token: ; value: ;
token: identifier value: while
token: (value: (
token: identifier value: pass
token:) value:)
token: Error value: {
token: Error value: }

Assumptions:

- The empty string was defined to be the character ‘\0’.
- Our state machines design is based on Moore design.
- The symbol table contains each identifier's value.
- If an input sequence led the machine to halt (don’t change its state) the last token found (last final state passed) is that obtained to perform the maximal munch. In the case that no final state was passed throughout this input sequence, the token is replaced with the statement “Error” and the corresponding unknown input value is shown.
- The usage of the cantor function to obtain a unique integer from two integers.
- The name of expression(token) or the definition contains only capital or small letters.

Phase II

DATA STRUCTURES :

Syntax Analyzer Model:

Struct production: This structure represents the productions as they are needed to be treated in keeping track of their first set to fill the parsing table correctly. They contain a vector of strings representing the RHS of the production and a vector of string representing their first set. We used a vector of strings to ease the manipulation instead of depending on separating the strings with spaces.

Struct non_terminal: This structure represents the non_terminals in the productions (LHS). We needed to keep data representing the non terminal as a separate entity from the production to be able to manipulate it, calculating the first and follow sets. It also helped in performing left factoring and removing the left recursions as will be shown. The struct contains the name of the non terminal, a vector of strings for the first set, a vector of strings for the follow set and a vector of productions representing the productions in which this non terminal is the LHS (separated by |).

Class syntax_analyzer: This class is used to perform the main operations in the syntax analysis. It is a singleton containing all the needed data and operations. It contains a map mapping each non terminal name to its corresponding nonterminal structure data, a vector of strings having the name of each non_terminal in the same order as that in the input file and a map mapping each nonterminal to a map which given a terminal input returns the corresponding RHS of the production (represents the parsing table). This class contains all the required utilities as adding a non terminal and manipulating it after parsing. It also contains the functions that write the results into the output file and functions that help in getting the parsing table entries according to the panic mode error recovery. The remaining functions are to be elaborated more later.

Parser Data Structures:

Struct nonTerminal_parser : it holds data about every non terminal read from file after applying the left factoring and eliminating the left recursion. It contains the name of every non terminal and vector of vectors of string that holds the productions of each one.

FUNCTIONS and ALGORITHMS :

Parser:

readfile_syn_rules() :

It reads every line from the syntax_rules file that starts with (‘#’) as a new non terminal and keeps concatenating all strings that start by (‘|’) to the previous non terminal and finally stores all these non terminals in a vector of struct nonTerminal_parser .

The right hand side of each non terminal is split by (‘|’) each one represents a certain production and each production is split by single or multiple spaces (‘ ’) each one represents either a non terminal or a terminal.

BONUS PART

Eliminate left recursion() :

The function starts by calling the vector of all non terminals read from the syntax_rules file. Then we look for each non terminal in the right hand side of each non terminal and compare it with the previous non terminals if they have the same names then we substitute the current non terminal with the productions of the previous one (that holds the same name) and reserving the rest of essential non terminal with each of newly added productions.

The next step is to eliminate the immediate left recursion by looking at the first string of each production of each non terminal if it has the same name as the current non terminal then we delete this non terminal from each production and keep the rest of production added to it the name of the non terminal + “ ‘ ” in a temporary vector and remove it from the essential non terminal.

Finally , we check if the temporary vector is empty then there is no immediate left recursion for the current terminal and continue to the next non terminal, else then we add to each production of the current non terminal the name of the non terminal + “ ‘ ” and create a new non terminal with the same name as the current non terminal + “ ‘ ” with the productions of the temporary vector.

left factoring() :

The main idea of left factoring is eliminating all productions of a nonterminal with the same prefix and creating a new nonterminal for these productions.

The algorithm starts by looping on nonterminals, one by one. At each nonterminal, we point at production, get the first symbol of this production and search for all following productions that start with the same symbol. If there is any production that satisfies this condition, we then eliminate it from the current nonterminal. At the end of this step, we also eliminate the current production at which we are currently pointing.

The next step is to find the longest prefix between these productions. We then create a new production with the longest prefix symbols appended with the symbol of the new nonterminal that will be created for eliminated productions and add it to the current nonterminal-which is the main loop nonterminal. Finally, we create a new nonterminal for all productions we have eliminated and add them to the newly-created nonterminal, removing the longest prefix symbols from each.

We repeat these steps for all productions of each nonterminal until there is not any production left.

Screenshots of the bonus part:

- **Not immediate left recursion**

```
# S = A a | b
# A = A c | S d | f
```

```
non terminal 1 : S
production 1 : A a
production 2 : b

non terminal 2 : A
production 1 : b d A'
production 2 : f A'

non terminal 3 : A'
production 1 : c A'
production 2 : a d A'
production 3 : \L
```

- **Immediate left recursion**

```
# S = a S | b X
# X = X X c | X d | Y
# Y = Y e | f | g
```

```
non terminal 1 : S
production 1 : a S
production 2 : b X

non terminal 2 : X
production 1 : Y X'

non terminal 3 : X'
production 1 : Y X' c X'
production 2 : d X'
production 3 : \L

non terminal 4 : Y
production 1 : f Y'
production 2 : g Y'

non terminal 5 : Y'
production 1 : e Y'
production 2 : \L
```

```
# E = E + T | T
# T = T * F | F
# F = id | ( E )
```

```
non terminal 1 : E
production 1 : T E'

non terminal 2 : E'
production 1 : + T E'
production 2 : \L

non terminal 3 : T
production 1 : F T'

non terminal 4 : T'
production 1 : * F T'
production 2 : \L

non terminal 5 : F
production 1 : id
production 2 : ( E )
```

- **Left factoring**

```
# A = a b B | a B | c d g | c d e B | c d f B
```

```
non terminal 1 : A
production 1 : a A1
production 2 : c d A2

non terminal 2 : A1
production 1 : b B
production 2 : B

non terminal 3 : A2
production 1 : g
production 2 : e B
production 3 : f B
```

```
# A = a d | a | a b | a b c | b
```

```
non terminal 1 : A
production 1 : a A1
production 2 : b

non terminal 2 : A1
production 1 : d
production 2 : \L
production 3 : b A11

non terminal 3 : A11
production 1 : \L
production 2 : c
```

Syntax Analyzer Model:

Algorithm to calculate first:

The first calculation is split into two main parts which are calculating the first of a single symbol (terminal or non terminal) and calculating the first of a vector of symbols (corresponding to the RHS of a production). The first production is done by some sort of recursion where the second part of the calculation is called in the first part and vice versa. The algorithm is as follows:

- The function **get_first_single** is called for each non terminal that does the following:
    - This function calculates the first of the given symbol after some processing and eventually stores it in the first of the corresponding non terminal structure.
    - If the given symbol is a terminal the function returns this terminal, if this is a non terminal and has its first already calculated the function just returns the first. These are the conditions to stop the recursion.
    - Otherwise, the first calculation algorithm is made, where for each production in which this non terminal is the LHS we get the first of the RHS and add it to the first of this non terminal.
- This is done by calling the **get_first_RHS** for each production that does the following:
    - If this RHS is only epsilon we just return epsilon as its first.
    - Otherwise, for each string in this RHS processed from left to right, we add the first of this string to the first of the productions without adding epsilon.
    - If epsilon is in the first of this string, we continue to the next string and process it similarly.
    - We get the first of each string symbol by calling the function get_first_single on this symbol that works as was shown.
    - If we passed all symbols without breaking (ie: they all have epsilon in their first sets) we add epsilon to the first set of the result.
- At the end we add the calculated RHS first to this production and the calculated non terminal first to this non terminal and the first calculation terminates having been made to all the non terminals (as it passes through all their productions it also calculates the first for those productions).

Functions to calculate follow:

set_follow() : Recursive function that implements the Follow algorithm.

The function takes as input the non-terminal symbol and a vector of pairs.
The vector is used to track the arguments sent to the function to make sure not to send a non-terminal that has been sent before and didn't get its follow set calculated, we can put this into simpler words that it prevents running into an infinite recursion.

The function returns a vector that contains the follow set of the non-terminal.

The algorithm calculates the follow set for every non-terminal symbol, this is done as following:-

- Simply we add a '$' symbol to the follow set in case the non-terminal is a start symbol.
- We loop over all the production rules escaping the ones that don't have the given non-terminal in its RHS.
- For the rest of the production rules, we do the following:-
    - If the given non-terminal is at the end of the rule then its follow set is the same as the non-terminal in the LHS.
    - Otherwise, the follow set is the first set of the following symbols excluding the epsilon.
    - If we found an epsilon in the first then we add the follow set of the non-terminal in the LHS.
- The function handles multiple appearances of the given non-terminal in the rule by going over the symbols and collecting the indices of where we found the non-terminal.
- The function checks at first if the follow set was calculated before or not and if so, we return it.

Functions to calculate parsing table:

fill_parsing_table() : Boolean function implements the parsing table algorithm.

The function takes the non-terminal symbol as input.

The function returns a boolean indicating a double entry in the table, false if a double entry occurred and true otherwise. In case of a double entry, an appropriate error message is printed.

The algorithm works as follows:

- We loop over all the production rules of the given non-terminal to fill in its row in the table.
- We get the first set of the production and set the entry with the production rule for each terminal in the first set, excluding the epsilon.
- If epsilon exists in the calculated first set then we set the entry of each terminal in the follow set of the non-terminal with the production.

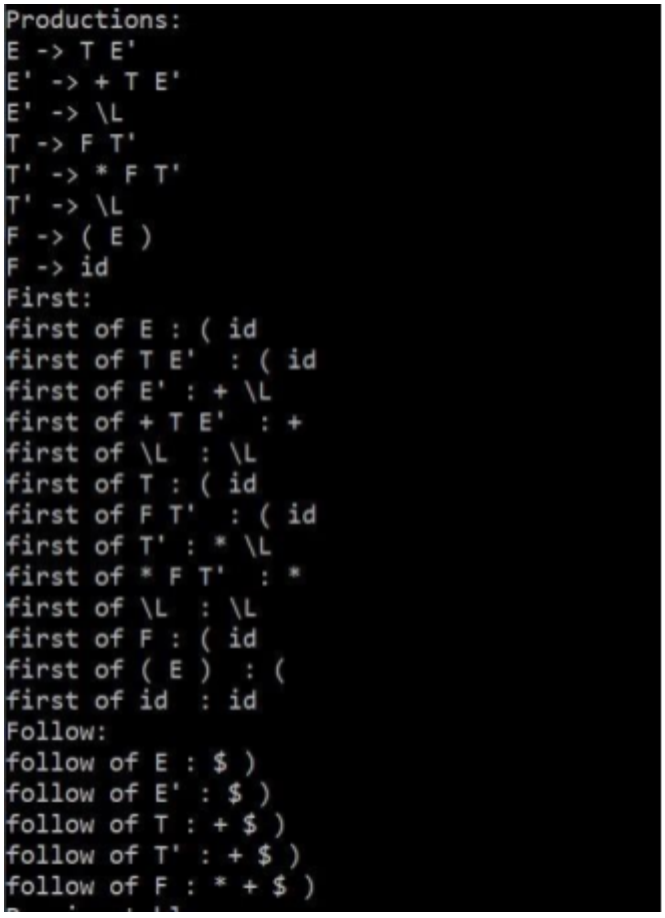
- Then we set the empty entries of the terminals in the follow set of the non-terminal with a synch entry.
- This function is called for all the non terminals after calculating the first and follow for each non terminal as was shown.

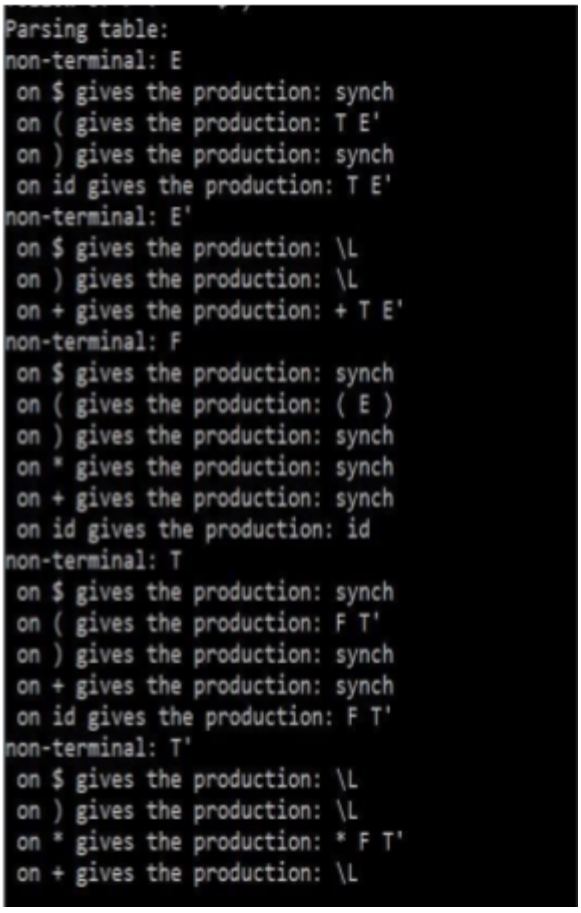
set_parsing_table() : Boolean function.

This is the main function that completely fills the parsing table. It first calculates the first and follow for all the non terminals and then calls **fill_parsing_table** for each non terminal. If in any iteration, **fill_parsing_table** returns false, the filling stops and the function returns false (not an LL(1) grammar), otherwise on successful completion of the filling, the function returns true.

Screenshots for the parsing table calculation:

Lecture examples:





```
Productions:
S -> A b S
S -> e
S -> \L
A -> a
A -> c A d

First:
first of S : a c e \L
first of A b S  : a c
first of e  : e
first of \L  : \L
first of A : a c
first of a  : a
first of c A d  : c

Follow:
follow of S : $
follow of A : b d

Parsing table:
non-terminal: A
 on a gives the production: a
 on b gives the production: synch
 on c gives the production: c A d
 on d gives the production: synch
non-terminal: S
 on $ gives the production: \L
 on a gives the production: A b S
 on c gives the production: A b S
 on e gives the production: e
```

```
Double entry at non terminal: S input: h productions: A C B  and C b B
Error: not LL(1) grammar.

Productions:
S -> A C B
S -> C b B
S -> B a
A -> d a
A -> B C
B -> g
B -> \L
C -> h
C -> \L

First:
first of S : d g h \L b a
first of A C B  : d g h \L
first of C b B  : h b
first of B a  : g a
first of A : d g h \L
first of d a  : d
first of B C  : g h \L
first of B : g \L
first of g  : g
first of \L  : \L
first of C : h \L
first of h  : h
first of \L  : \L

Follow:
follow of S : $
follow of A : h g $
follow of B : $ a h g
follow of C : g $ b h
```

Sheet4 examples:

```
Productions:
S -> R T
R -> s U R b
R -> \L
U -> u U
U -> \L
V -> v V
V -> \L
T -> V t T
T -> \L

First:
first of S : s v t \L
first of R T  : s v t \L
first of R : s \L
first of s U R b  : s
first of \L  : \L
first of U : u \L
first of u U  : u
first of \L  : \L
first of V : v \L
first of v V  : v
first of \L  : \L
first of T : v t \L
first of V t T  : v t
first of \L  : \L
```

```
Follow:
follow of S : $
follow of R : v t $ b
follow of U : s b
follow of V : t
follow of T : $
```

```
Parsing table:
non-terminal: R
 on $ gives the production: \L
 on b gives the production: \L
 on s gives the production: s U R b
 on t gives the production: \L
 on v gives the production: \L
non-terminal: S
 on $ gives the production: R T
 on s gives the production: R T
 on t gives the production: R T
 on v gives the production: R T
non-terminal: T
 on $ gives the production: \L
 on t gives the production: V t T
 on v gives the production: V t T
non-terminal: U
 on b gives the production: \L
 on s gives the production: \L
 on u gives the production: u U
non-terminal: V
 on t gives the production: \L
 on v gives the production: v V
```

```
Productions:
S -> a S
S -> b X
X -> Y X'
X' -> X c X'
X' -> d X'
X' -> \L
Y -> f Y'
Y -> g Y'
Y' -> e Y'
Y' -> \L

First:
first of S : a b
first of a S  : a
first of b X  : b
first of X : f g
first of Y X'  : f g
first of X' : f g d \L
first of X c X'  : f g
first of d X'  : d
first of \L  : \L
first of Y : f g
first of f Y'  : f
first of g Y'  : g
first of Y' : e \L
first of e Y'  : e
first of \L  : \L
```

```
Follow:
follow of S : $
follow of X : $ c
follow of X' : $ c
follow of Y : f g d $ c
follow of Y' : f g d $ c
```

```
Parsing table:
non-terminal: S
 on $ gives the production: synch
 on a gives the production: a S
 on b gives the production: b X
non-terminal: X
 on $ gives the production: synch
 on c gives the production: synch
 on f gives the production: Y X'
 on g gives the production: Y X'
non-terminal: X'
 on $ gives the production: \L
 on c gives the production: \L
 on d gives the production: d X'
 on f gives the production: X c X'
 on g gives the production: X c X'
non-terminal: Y
 on $ gives the production: synch
 on c gives the production: synch
 on d gives the production: synch
 on f gives the production: f Y'
 on g gives the production: g Y'
non-terminal: Y'
 on $ gives the production: \L
 on c gives the production: \L
 on d gives the production: \L
 on e gives the production: e Y'
 on f gives the production: \L
 on g gives the production: \L
```

Sheet 5 examples:

```
Productions:
bexpr -> bterm bexpr'
bexpr' -> or bterm bexpr'
bexpr' -> \L
bterm -> bfactor bterm'
bterm' -> and bfactor bterm'
bterm' -> \L
bfactor -> not bfactor
bfactor -> ( bexpr )
bfactor -> true
bfactor -> false

First:
first of bexpr : not ( true false
first of bterm bexpr'  : not ( true false
first of bexpr' : or \L
first of or bterm bexpr'  : or
first of \L  : \L
first of bterm : not ( true false
first of bfactor bterm'  : not ( true false
first of bterm' : and \L
first of and bfactor bterm'  : and
first of \L  : \L
first of bfactor : not ( true false
first of not bfactor  : not
first of ( bexpr )  : (
first of true  : true
first of false  : false
```

```
Follow:
follow of bexpr : $ )
follow of bexpr' : $ )
follow of bterm : or $ )
follow of bterm' : or $ )
follow of bfactor : and or $ )
```

```
Parsing table:
non-terminal: bexpr
 on $ gives the production: synch
 on ( gives the production: bterm bexpr'
 on ) gives the production: synch
 on false gives the production: bterm bexpr'
 on not gives the production: bterm bexpr'
 on true gives the production: bterm bexpr'
non-terminal: bexpr'
 on $ gives the production: \L
 on ) gives the production: \L
 on or gives the production: or bterm bexpr'
non-terminal: bfactor
 on $ gives the production: synch
 on ( gives the production: ( bexpr )
 on ) gives the production: synch
 on and gives the production: synch
 on false gives the production: false
 on not gives the production: not bfactor
 on or gives the production: synch
 on true gives the production: true
non-terminal: bterm
 on $ gives the production: synch
 on ( gives the production: bfactor bterm'
 on ) gives the production: synch
 on false gives the production: bfactor bterm'
 on not gives the production: bfactor bterm'
 on or gives the production: synch
 on true gives the production: bfactor bterm'
```

```
non-terminal: bterm'
 on $ gives the production: \L
 on ) gives the production: \L
 on and gives the production: and bfactor bterm'
 on or gives the production: \L
```

Other examples:

```
Productions:
S -> A a A b
S -> B b B a
A -> \L
B -> \L

First:
first of S : a b
first of A a A b  : a
first of B b B a  : b
first of A : \L
first of \L  : \L
first of B : \L
first of \L  : \L

Follow:
follow of S : $
follow of A : a b
follow of B : b a

Parsing table:
non-terminal: A
 on a gives the production: \L
 on b gives the production: \L
non-terminal: B
 on a gives the production: \L
 on b gives the production: \L
non-terminal: S
 on $ gives the production: synch
 on a gives the production: A a A b
 on b gives the production: B b B a
```

```
Productions:
S -> ( L )
S -> a
L -> S L'
L' -> , S L'
L' -> \L

First:
first of S : ( a
first of ( L )  : (
first of a  : a
first of L : ( a
first of S L'  : ( a
first of L' : , \L
first of , S L'  : ,
first of \L  : \L

Follow:
follow of S : $ , )
follow of L : )
follow of L' : )

Parsing table:
non-terminal: L
 on ( gives the production: S L'
 on ) gives the production: synch
 on a gives the production: S L'
non-terminal: L'
 on ) gives the production: \L
 on , gives the production: , S L'
non-terminal: S
 on $ gives the production: synch
 on ( gives the production: ( L )
 on ) gives the production: synch
 on , gives the production: synch
 on a gives the production: a
```

```
Productions:
S -> ( L )
S -> a
L -> S L'
L' -> , S L'
L' -> \L

First:
first of S : ( a
first of ( L )  : (
first of a  : a
first of L : ( a
first of S L'  : ( a
first of L' : , \L
first of , S L'  : ,
first of \L  : \L

Follow:
follow of S : $ , )
follow of L : )
follow of L' : )

Parsing table:
non-terminal: L
 on ( gives the production: S L'
 on ) gives the production: synch
 on a gives the production: S L'
non-terminal: L'
 on ) gives the production: \L
 on , gives the production: , S L'
non-terminal: S
 on $ gives the production: synch
 on ( gives the production: ( L )
 on ) gives the production: synch
 on , gives the production: synch
 on a gives the production: a
```

```
double entry at non terminal: S input: h productions: A C B  and C b B
Error: not LL(1) grammar.

Productions:
S -> A C B
S -> C b B
S -> B a
A -> d a
A -> B C
B -> g
B -> \L
C -> h
C -> \L

First:
first of S : d g h \L b a
first of A C B  : d g h \L
first of C b B  : h b
first of B a  : g a
first of A : d g h \L
first of d a  : d
first of B C  : g h \L
first of B : g \L
first of g  : g
first of \L  : \L
first of C : h \L
first of h  : h
first of \L  : \L

Follow:
follow of S : $
follow of A : h g $
follow of B : $ a h g
follow of C : g $ b h
```

Stack:

run_stack():

It starts by getting the vector of strings that holds the tokens that obtained from lexical analyzer after removing all error tokens. We push the (‘$’) which indicates that the stack is empty followed by the name of the first non terminal according to the input syntax rules. We compare the top of the stack with the current input string . The comparison results in one of the following scenarios:

- If the top of stack is terminal and equals to the current input then we pop this terminal from the stack and remove it from the input
- If the top of stack is terminal but is not equal to the current input then we only pop the terminal from the stack and keep it in the input

- If the top of stack is non terminal then we need to look for the entry in the parsing table that holds the production of this non terminal under the current input.
    1. If the entry is empty then we discard the input (remove the current input).
    2. If the entry is synch then we only pop the non terminal from the stack .
    3. If the entry contains a production then we pop the current non terminal from the stack and push the corresponding production.

We keep repeating the previous steps until the stack becomes empty (contains (‘$’) only) then we compare it with the remaining input if it also contains only (‘$’) then the input matches the grammar , else the input doesn’t match the grammar.

SCREEN SHOTS :

- **Example in project PDF:**

```
# 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 = '+' | '-'
```

```
First:
first of METHOD_BODY : int float if while id
first of STATEMENT_LIST  : int float if while id
first of STATEMENT_LIST : int float if while id
first of STATEMENT STATEMENT_LIST'  : int float if while id
first of STATEMENT_LIST' : int float if while id \L
first of STATEMENT STATEMENT_LIST'  : int float if while id
first of \L  : \L
first of STATEMENT : int float if while id
first of DECLARATION  : int float
first of IF  : if
first of WHILE  : while
first of ASSIGNMENT  : id
first of DECLARATION : int float
first of PRIMITIVE_TYPE id ;  : int float
first of PRIMITIVE_TYPE : int float
first of int  : int
first of float  : float
first of IF : if
first of if ( EXPRESSION ) { STATEMENT } else { STATEMENT }  : if
first of WHILE : while
first of while ( EXPRESSION ) { STATEMENT }  : while
first of ASSIGNMENT : id
first of id = EXPRESSION ;  : id
first of EXPRESSION : id num ( + -
first of SIMPLE_EXPRESSION EXPRESSION1  : id num ( + -
first of SIMPLE_EXPRESSION : id num ( + -
first of TERM SIMPLE_EXPRESSION'  : id num (
first of SIGN TERM SIMPLE_EXPRESSION'  : + -
first of SIMPLE_EXPRESSION' : addop \L
first of addop TERM SIMPLE_EXPRESSION'  : addop
first of \L  : \L
first of TERM : id num (
first of FACTOR TERM'  : id num (
first of TERM' : mulop \L
first of mulop FACTOR TERM'  : mulop
first of \L  : \L
first of FACTOR : id num (
first of id  : id
first of num  : num
first of ( EXPRESSION )  : (
first of SIGN : + -
first of +  : +
```

```
first of -  : -
first of EXPRESSION1 : \L relop
first of \L  : \L
first of relop SIMPLE_EXPRESSION  : relop


Follow:
follow of METHOD_BODY : $
follow of STATEMENT_LIST : $
follow of STATEMENT_LIST' : $
follow of STATEMENT : int float if while id $ }
follow of DECLARATION : int float if while id $ }
follow of PRIMITIVE_TYPE : id
follow of IF : int float if while id $ }
follow of WHILE : int float if while id $ }
follow of ASSIGNMENT : int float if while id $ }
follow of EXPRESSION : ) ;
follow of SIMPLE_EXPRESSION : relop ) ;
follow of SIMPLE_EXPRESSION' : relop ) ;
follow of TERM : addop relop ) ;
follow of TERM' : addop relop ) ;
follow of FACTOR : mulop addop relop ) ;
follow of SIGN : id num (
follow of EXPRESSION1 : ) ;
```

```
Parsing table:
non-terminal: ASSIGNMENT
 on $ gives the production: synch
 on float gives the production: synch
 on id gives the production: id = EXPRESSION ;
 on if gives the production: synch
 on int gives the production: synch
 on while gives the production: synch
 on } gives the production: synch
non-terminal: DECLARATION
 on $ gives the production: synch
 on float gives the production: PRIMITIVE_TYPE id ;
 on id gives the production: synch
 on if gives the production: synch
 on int gives the production: PRIMITIVE_TYPE id ;
 on while gives the production: synch
 on } gives the production: synch
non-terminal: EXPRESSION
 on ( gives the production: SIMPLE_EXPRESSION EXPRESSION1
 on ) gives the production: synch
 on + gives the production: SIMPLE_EXPRESSION EXPRESSION1
 on - gives the production: SIMPLE_EXPRESSION EXPRESSION1
 on ; gives the production: synch
 on id gives the production: SIMPLE_EXPRESSION EXPRESSION1
 on num gives the production: SIMPLE_EXPRESSION EXPRESSION1
non-terminal: EXPRESSION1
 on ) gives the production: \L
 on ; gives the production: \L
 on relop gives the production: relop SIMPLE_EXPRESSION
non-terminal: FACTOR
 on ( gives the production: ( EXPRESSION )
 on ) gives the production: synch
 on ; gives the production: synch
 on addop gives the production: synch
 on id gives the production: id
 on mulop gives the production: synch
 on num gives the production: num
 on relop gives the production: synch
```

```
non-terminal: IF
 on $ gives the production: synch
 on float gives the production: synch
 on id gives the production: synch
 on if gives the production: if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
 on int gives the production: synch
 on while gives the production: synch
 on } gives the production: synch
non-terminal: METHOD_BODY
 on $ gives the production: synch
 on float gives the production: STATEMENT_LIST
 on id gives the production: STATEMENT_LIST
 on if gives the production: STATEMENT_LIST
 on int gives the production: STATEMENT_LIST
 on while gives the production: STATEMENT_LIST
non-terminal: PRIMITIVE_TYPE
 on float gives the production: float
 on id gives the production: synch
 on int gives the production: int
non-terminal: SIGN
 on ( gives the production: synch
 on + gives the production: +
 on - gives the production: -
 on id gives the production: synch
 on num gives the production: synch
non-terminal: SIMPLE_EXPRESSION
 on ( gives the production: TERM SIMPLE_EXPRESSION'
 on ) gives the production: synch
 on + gives the production: SIGN TERM SIMPLE_EXPRESSION'
 on - gives the production: SIGN TERM SIMPLE_EXPRESSION'
 on ; gives the production: synch
 on id gives the production: TERM SIMPLE_EXPRESSION'
 on num gives the production: TERM SIMPLE_EXPRESSION'
 on relop gives the production: synch
non-terminal: SIMPLE_EXPRESSION'
 on ) gives the production: \L
 on ; gives the production: \L
 on addop gives the production: addop TERM SIMPLE_EXPRESSION'
 on relop gives the production: \L
```

```
non-terminal: STATEMENT
 on $ gives the production: synch
 on float gives the production: DECLARATION
 on id gives the production: ASSIGNMENT
 on if gives the production: IF
 on int gives the production: DECLARATION
 on while gives the production: WHILE
 on } gives the production: synch
non-terminal: STATEMENT_LIST
 on $ gives the production: synch
 on float gives the production: STATEMENT STATEMENT_LIST'
 on id gives the production: STATEMENT STATEMENT_LIST'
 on if gives the production: STATEMENT STATEMENT_LIST'
 on int gives the production: STATEMENT STATEMENT_LIST'
 on while gives the production: STATEMENT STATEMENT_LIST'
non-terminal: STATEMENT_LIST'
 on $ gives the production: \L
 on float gives the production: STATEMENT STATEMENT_LIST'
 on id gives the production: STATEMENT STATEMENT_LIST'
 on if gives the production: STATEMENT STATEMENT_LIST'
 on int gives the production: STATEMENT STATEMENT_LIST'
 on while gives the production: STATEMENT STATEMENT_LIST'
non-terminal: TERM
 on ( gives the production: FACTOR TERM'
 on ) gives the production: synch
 on ; gives the production: synch
 on addop gives the production: synch
 on id gives the production: FACTOR TERM'
 on num gives the production: FACTOR TERM'
 on relop gives the production: synch
non-terminal: TERM'
 on ) gives the production: \L
 on ; gives the production: \L
 on addop gives the production: \L
 on mulop gives the production: mulop FACTOR TERM'
 on relop gives the production: \L
```

```
non-terminal: WHILE
 on $ gives the production: synch
 on float gives the production: synch
 on id gives the production: synch
 on if gives the production: synch
 on int gives the production: synch
 on while gives the production: while ( EXPRESSION ) { STATEMENT }
 on } gives the production: synch
```

Test for input 1:

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

Tokens of lexical :

```
Lexical Output:
token: int value: int
token: id value: sum
token: , value: ,
token: id value: count
token: , value: ,
token: id value: pass
token: , value: ,
token: id value: mnt
token: ; value: ;
token: while value: while
token: ( value: (
token: id value: pass
token: relop value: !=
token: num value: 10
token: ) value: )
token: { value: {
token: id value: pass
token: assign value: =
token: id value: pass
token: addop value: +
token: num value: 1
token: ; value: ;
token: } value: }

Symbols in symbol table:
1 : mnt
2 : pass
3 : count
4 : sum
```

The stack :

```
METHOD_BODY
METHOD_BODY----->STATEMENT_LIST
STATEMENT_LIST
STATEMENT_LIST----->STATEMENT  STATEMENT_LIST'
STATEMENT  STATEMENT_LIST'
STATEMENT----->DECLARATION
DECLARATION  STATEMENT_LIST'
DECLARATION----->PRIMITIVE_TYPE  id  ;
PRIMITIVE_TYPE  id  ;  STATEMENT_LIST'
PRIMITIVE_TYPE----->int
int  id  ;  STATEMENT_LIST'
terminals match
int  id  ;  STATEMENT_LIST'
terminals match
int  id  ;  STATEMENT_LIST'
terminals mismatch(pop from stack)
int  id  STATEMENT_LIST'
error (remove from input)
int  id  STATEMENT_LIST'
STATEMENT_LIST'----->STATEMENT  STATEMENT_LIST'
int  id  STATEMENT  STATEMENT_LIST'
STATEMENT----->ASSIGNMENT
int  id  ASSIGNMENT  STATEMENT_LIST'
ASSIGNMENT----->id  assign  EXPRESSION  ;
int  id  id  assign  EXPRESSION  ;  STATEMENT_LIST'
terminals match
int  id  id  assign  EXPRESSION  ;  STATEMENT_LIST'
terminals mismatch(pop from stack)
int  id  id  EXPRESSION  ;  STATEMENT_LIST'
error (remove from input)
int  id  id  EXPRESSION  ;  STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION  EXPRESSION1
int  id  id  SIMPLE_EXPRESSION  EXPRESSION1  ;  STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM  SIMPLE_EXPRESSION'
int  id  id  TERM  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
TERM----->FACTOR  TERM'
int  id  id  FACTOR  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
FACTOR----->id
int  id  id  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
terminals match
int  id  id  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
error (remove from input)
int  id  id  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
```

```
error (remove from input)
int  id  id  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
TERM'----->\L
int  id  id  id  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int  id  id  id  EXPRESSION1  ;  STATEMENT_LIST'
EXPRESSION1----->\L
int  id  id  id  ;  STATEMENT_LIST'
terminals match
int  id  id  id  ;  STATEMENT_LIST'
STATEMENT_LIST'----->STATEMENT  STATEMENT_LIST'
int  id  id  id  ;  STATEMENT  STATEMENT_LIST'
STATEMENT----->WHILE
int  id  id  id  ;  WHILE  STATEMENT_LIST'
WHILE----->while  (  EXPRESSION  )  {  STATEMENT  }
int  id  id  id  ;  while  (  EXPRESSION  )  {  STATEMENT  }  STATEMENT_LIST'
terminals match
int  id  id  id  ;  while  (  EXPRESSION  )  {  STATEMENT  }  STATEMENT_LIST'
terminals match
int  id  id  id  ;  while  (  EXPRESSION  )  {  STATEMENT  }  STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION  EXPRESSION1
int  id  id  id  ;  while  (  SIMPLE_EXPRESSION  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM  SIMPLE_EXPRESSION'
int  id  id  id  ;  while  (  TERM  SIMPLE_EXPRESSION'  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
TERM----->FACTOR  TERM'
int  id  id  id  ;  while  (  FACTOR  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
FACTOR----->id
int  id  id  id  ;  while  (  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
terminals match
int  id  id  id  ;  while  (  id  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
TERM'----->\L
int  id  id  id  ;  while  (  id  SIMPLE_EXPRESSION'  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int  id  id  id  ;  while  (  id  EXPRESSION1  )  {  STATEMENT  }  STATEMENT_LIST'
EXPRESSION1----->relop  SIMPLE_EXPRESSION
int  id  id  id  ;  while  (  id  relop  SIMPLE_EXPRESSION  )  {  STATEMENT  }  STATEMENT_LIST'
terminals match
int  id  id  id  ;  while  (  id  relop  SIMPLE_EXPRESSION  )  {  STATEMENT  }  STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM  SIMPLE_EXPRESSION'
int  id  id  id  ;  while  (  id  relop  TERM  SIMPLE_EXPRESSION'  )  {  STATEMENT  }  STATEMENT_LIST'
TERM----->FACTOR  TERM'
int  id  id  id  ;  while  (  id  relop  FACTOR  TERM'  SIMPLE_EXPRESSION'  )  {  STATEMENT  }  STATEMENT_LIST'
```

```
FACTOR----->num
int id id id ; while ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } STATEMENT_LIST'
TERM'----->\L
int id id id ; while ( id relop num SIMPLE_EXPRESSION' ) { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int id id id ; while ( id relop num ) { STATEMENT } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { STATEMENT } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { STATEMENT } STATEMENT_LIST'
STATEMENT----->ASSIGNMENT
int id id id ; while ( id relop num ) { ASSIGNMENT } STATEMENT_LIST'
ASSIGNMENT----->id assign EXPRESSION ;
int id id id ; while ( id relop num ) { id assign EXPRESSION ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign EXPRESSION ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign EXPRESSION ; } STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION EXPRESSION1
int id id id ; while ( id relop num ) { id assign SIMPLE_EXPRESSION EXPRESSION1 ; } STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM SIMPLE_EXPRESSION'
int id id id ; while ( id relop num ) { id assign TERM SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
TERM----->FACTOR TERM'
int id id id ; while ( id relop num ) { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
FACTOR----->id
int id id id ; while ( id relop num ) { id assign id TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign id TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
TERM'----->\L
int id id id ; while ( id relop num ) { id assign id SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->addop TERM SIMPLE_EXPRESSION'
int id id id ; while ( id relop num ) { id assign id addop TERM SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign id addop TERM SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
TERM----->FACTOR TERM'
int id id id ; while ( id relop num ) { id assign id addop FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
FACTOR----->num
int id id id ; while ( id relop num ) { id assign id addop num TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign id addop num TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
```

```
TERM'----->\L
int id id id ; while ( id relop num ) { id assign id addop num SIMPLE_EXPRESSION' EXPRESSION1 ; } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int id id id ; while ( id relop num ) { id assign id addop num EXPRESSION1 ; } STATEMENT_LIST'
EXPRESSION1----->\L
int id id id ; while ( id relop num ) { id assign id addop num ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign id addop num ; } STATEMENT_LIST'
terminals match
int id id id ; while ( id relop num ) { id assign id addop num ; } STATEMENT_LIST'
STATEMENT_LIST'----->\L
int id id id ; while ( id relop num ) { id assign id addop num ; }
input matches the grammar with errors
```

Test for input 2:

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

Tokens of lexical:

```
Lexical Output:
token: int value: int
token: id value: x
token: ; value: ;
token: id value: x
token: assign value: =
token: num value: 5
token: ; value: ;
token: if value: if
token: ( value: (
token: id value: x
token: relop value: >
token: num value: 2
token: ) value: )
token: { value: {
token: id value: x
token: assign value: =
token: num value: 0
token: ; value: ;
token: } value: }

Symbols in symbol table:
1 : x
```

The stack:

```
METHOD_BODY
METHOD_BODY----->STATEMENT_LIST
STATEMENT_LIST
STATEMENT_LIST----->STATEMENT  STATEMENT_LIST'
STATEMENT  STATEMENT_LIST'
STATEMENT----->DECLARATION
DECLARATION  STATEMENT_LIST'
DECLARATION----->PRIMITIVE_TYPE  id  ;
PRIMITIVE_TYPE  id  ;  STATEMENT_LIST'
PRIMITIVE_TYPE----->int
int  id  ;  STATEMENT_LIST'
terminals match
int  id  ;  STATEMENT_LIST'
terminals match
int  id  ;  STATEMENT_LIST'
terminals match
int  id  ;  STATEMENT_LIST'
STATEMENT_LIST'----->STATEMENT  STATEMENT_LIST'
int  id  ;  STATEMENT  STATEMENT_LIST'
STATEMENT----->ASSIGNMENT
int  id  ;  ASSIGNMENT  STATEMENT_LIST'
ASSIGNMENT----->id  assign  EXPRESSION  ;
int  id  ;  id  assign  EXPRESSION  ;  STATEMENT_LIST'
terminals match
int  id  ;  id  assign  EXPRESSION  ;  STATEMENT_LIST'
terminals match
int  id  ;  id  assign  EXPRESSION  ;  STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION  EXPRESSION1
int  id  ;  id  assign  SIMPLE_EXPRESSION  EXPRESSION1  ;  STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM  SIMPLE_EXPRESSION'
int  id  ;  id  assign  TERM  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
TERM----->FACTOR  TERM'
int  id  ;  id  assign  FACTOR  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
FACTOR----->num
int  id  ;  id  assign  num  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
terminals match
int  id  ;  id  assign  num  TERM'  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
TERM'----->\L
int  id  ;  id  assign  num  SIMPLE_EXPRESSION'  EXPRESSION1  ;  STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int  id  ;  id  assign  num  EXPRESSION1  ;  STATEMENT_LIST'
EXPRESSION1----->\L
int  id  ;  id  assign  num  ;  STATEMENT_LIST'
```

```
terminals match
int id ; id assign num ; STATEMENT_LIST'
STATEMENT_LIST'----->STATEMENT STATEMENT_LIST'
int id ; id assign num ; STATEMENT STATEMENT_LIST'
STATEMENT----->IF
int id ; id assign num ; IF STATEMENT_LIST'
IF----->if ( EXPRESSION ) { STATEMENT } else { STATEMENT }
int id ; id assign num ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION EXPRESSION1
int id ; id assign num ; if ( SIMPLE_EXPRESSION EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM SIMPLE_EXPRESSION'
int id ; id assign num ; if ( TERM SIMPLE_EXPRESSION' EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
TERM----->FACTOR TERM'
int id ; id assign num ; if ( FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
FACTOR----->id
int id ; id assign num ; if ( id TERM' SIMPLE_EXPRESSION' EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id TERM' SIMPLE_EXPRESSION' EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
TERM'----->\L
int id ; id assign num ; if ( id SIMPLE_EXPRESSION' EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int id ; id assign num ; if ( id EXPRESSION1 ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
EXPRESSION1----->relop SIMPLE_EXPRESSION
int id ; id assign num ; if ( id relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop SIMPLE_EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM SIMPLE_EXPRESSION'
int id ; id assign num ; if ( id relop TERM SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
TERM----->FACTOR TERM'
int id ; id assign num ; if ( id relop FACTOR TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
FACTOR----->num
int id ; id assign num ; if ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num TERM' SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
TERM'----->\L
int id ; id assign num ; if ( id relop num SIMPLE_EXPRESSION' ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int id ; id assign num ; if ( id relop num ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
```

```
terminals match
int id ; id assign num ; if ( id relop num ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { STATEMENT } else { STATEMENT } STATEMENT_LIST'
STATEMENT----->ASSIGNMENT
int id ; id assign num ; if ( id relop num ) { ASSIGNMENT } else { STATEMENT } STATEMENT_LIST'
ASSIGNMENT----->id assign EXPRESSION ;
int id ; id assign num ; if ( id relop num ) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { id assign EXPRESSION ; } else { STATEMENT } STATEMENT_LIST'
EXPRESSION----->SIMPLE_EXPRESSION  EXPRESSION1
int id ; id assign num ; if ( id relop num ) { id assign SIMPLE_EXPRESSION EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION----->TERM  SIMPLE_EXPRESSION'
int id ; id assign num ; if ( id relop num ) { id assign TERM SIMPLE_EXPRESSION' EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
TERM----->FACTOR  TERM'
int id ; id assign num ; if ( id relop num ) { id assign FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
FACTOR----->num
int id ; id assign num ; if ( id relop num ) { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { id assign num TERM' SIMPLE_EXPRESSION' EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
TERM'----->\L
int id ; id assign num ; if ( id relop num ) { id assign num SIMPLE_EXPRESSION' EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
SIMPLE_EXPRESSION'----->\L
int id ; id assign num ; if ( id relop num ) { id assign num EXPRESSION1 ; } else { STATEMENT } STATEMENT_LIST'
EXPRESSION1----->\L
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { STATEMENT } STATEMENT_LIST'
terminals match
int id ; id assign num ; if ( id relop num ) { id assign num ; } else { STATEMENT } STATEMENT_LIST'
terminals mismatch(pop from stack)
int id ; id assign num ; if ( id relop num ) { id assign num ; } { STATEMENT } STATEMENT_LIST'
terminals mismatch(pop from stack)
int id ; id assign num ; if ( id relop num ) { id assign num ; } STATEMENT } STATEMENT_LIST'
error (synch entry pop from stack)
int id ; id assign num ; if ( id relop num ) { id assign num ; } } STATEMENT_LIST'
terminals mismatch(pop from stack)
int id ; id assign num ; if ( id relop num ) { id assign num ; } STATEMENT_LIST'
STATEMENT_LIST'----->\L
int id ; id assign num ; if ( id relop num ) { id assign num ; }
input matches the grammar with errors
```

- **Example 2:**

```
# E = T E'
# E' = + T E' | \L
# T = F T'
# T' = * F T' | \L
# F = ( E )  | id
```

```
First:
first of E : ( id
first of T E'  : ( id
first of E' : + \L
first of + T E'  : +
first of \L  : \L
first of T : ( id
first of F T'  : ( id
first of T' : * \L
first of * F T'  : *
first of \L  : \L
first of F : ( id
first of ( E )  : (
first of id  : id

Follow:
follow of E : $ )
follow of E' : $ )
follow of T : + $ )
follow of T' : + $ )
follow of F : * + $ )
```

```
Parsing table:
non-terminal: E
 on $ gives the production: synch
 on ( gives the production: T E'
 on ) gives the production: synch
 on id gives the production: T E'
non-terminal: E'
 on $ gives the production: \L
 on ) gives the production: \L
 on + gives the production: + T E'
non-terminal: F
 on $ gives the production: synch
 on ( gives the production: ( E )
 on ) gives the production: synch
 on * gives the production: synch
 on + gives the production: synch
 on id gives the production: id
non-terminal: T
 on $ gives the production: synch
 on ( gives the production: F T'
 on ) gives the production: synch
 on + gives the production: synch
 on id gives the production: F T'
non-terminal: T'
 on $ gives the production: \L
 on ) gives the production: \L
 on * gives the production: * F T'
 on + gives the production: \L
```

Test for input 1 :

```
( id + ( * id )
```

Tokens of lexical :

```
Lexical Output:
token: ( value: (
token: id value: id
token: + value: +
token: ( value: (
token: * value: *
token: id value: id
token: ) value: )
```

The stack :

```
E
E----->T  E'
T  E'
T----->F  T'
F  T'  E'
F----->(  E  )
(  E  )  T'  E'
terminals match
(  E  )  T'  E'
E----->T  E'
(  T  E'  )  T'  E'
T----->F  T'
(  F  T'  E'  )  T'  E'
F----->id
(  id  T'  E'  )  T'  E'
terminals match
(  id  T'  E'  )  T'  E'
T'----->\L
(  id  E'  )  T'  E'
E'----->+  T  E'
(  id  +  T  E'  )  T'  E'
terminals match
(  id  +  T  E'  )  T'  E'
T----->F  T'
(  id  +  F  T'  E'  )  T'  E'
F----->(  E  )
(  id  +  (  E  )  T'  E'  )  T'  E'
terminals match
(  id  +  (  E  )  T'  E'  )  T'  E'
error (remove from input)
(  id  +  (  E  )  T'  E'  )  T'  E'
E----->T  E'
(  id  +  (  T  E'  )  T'  E'  )  T'  E'
T----->F  T'
(  id  +  (  F  T'  E'  )  T'  E'  )  T'  E'
F----->id
(  id  +  (  id  T'  E'  )  T'  E'  )  T'  E'
terminals match
(  id  +  (  id  T'  E'  )  T'  E'  )  T'  E'
T'----->\L
(  id  +  (  id  E'  )  T'  E'  )  T'  E'
E'----->\L
(  id  +  (  id  )  T'  E'  )  T'  E'
```

```
terminals match
(  id  +  (  id  )  T'  E'  )  T'  E'
T'----->\L
(  id  +  (  id  )  E'  )  T'  E'
E'----->\L
(  id  +  (  id  )  )  T'  E'
terminals mismatch(pop from stack)
(  id  +  (  id  )  T'  E'
T'----->\L
(  id  +  (  id  )  E'
E'----->\L
(  id  +  (  id  )
input matches the grammar with errors
```

Test for input 2 :

```
* + id ) + ( id *
```

Tokens of lexical :

```
Lexical Output:
token: * value: *
token: + value: +
token: id value: id
token: ) value: )
token: + value: +
token: ( value: (
token: id value: id
token: * value: *
```

The stack :

```
E
error (remove from input)
E
error (remove from input)
E
E----->T  E'
T  E'
T----->F  T'
F  T'  E'
F----->id
id  T'  E'
terminals match
id  T'  E'
T'----->\L
id  E'
E'----->\L
id
input mismatch the grammar
```

- **Example 3 :**

```
# S = R T
# R = s U R b | \L
# U = u U | \L
# V = v V | \L
# T = V t T | \L
```

```
First:
first of S : s v t \L
first of R T  : s v t \L
first of R : s \L
first of s U R b  : s
first of \L  : \L
first of U : u \L
first of u U  : u
first of \L  : \L
first of V : v \L
first of v V  : v
first of \L  : \L
first of T : v t \L
first of v V t T  : v
first of \L T1  : t \L
first of T1 : t \L
first of t T  : t
first of \L  : \L

Follow:
follow of S : $
follow of R : v t $ b
follow of U : s b
follow of V : t
follow of T : $
follow of T1 : $
```

```
Parsing table:
non-terminal: R
 on $ gives the production: \L
 on b gives the production: \L
 on s gives the production: s U R b
 on t gives the production: \L
 on v gives the production: \L
non-terminal: S
 on $ gives the production: R T
 on s gives the production: R T
 on t gives the production: R T
 on v gives the production: R T
non-terminal: T
 on $ gives the production: \L T1
 on t gives the production: \L T1
 on v gives the production: v V t T
non-terminal: T1
 on $ gives the production: \L
 on t gives the production: t T
non-terminal: U
 on b gives the production: \L
 on s gives the production: \L
 on u gives the production: u U
non-terminal: V
 on t gives the production: \L
 on v gives the production: v V
```

For input :

```
s s u u b b t v t
```

Tokens of lexical :

```
Lexical Output:
token: s value: s
token: s value: s
token: u value: u
token: u value: u
token: b value: b
token: b value: b
token: t value: t
token: v value: v
token: t value: t
```

The stack :

```
S
S----->R  T
R  T
R----->s  U  R  b
s  U  R  b  T
terminals match
s  U  R  b  T
U----->\L
s  R  b  T
R----->s  U  R  b
s  s  U  R  b  b  T
terminals match
s  s  U  R  b  b  T
U----->u  U
s  s  u  U  R  b  b  T
terminals match
s  s  u  U  R  b  b  T
U----->u  U
s  s  u  u  U  R  b  b  T
terminals match
s  s  u  u  U  R  b  b  T
U----->\L
s  s  u  u  R  b  b  T
R----->\L
s  s  u  u  b  b  T
terminals match
s  s  u  u  b  b  T
terminals match
s  s  u  u  b  b  T
T----->\L  T1
s  s  u  u  b  b  \L  T1
terminals mismatch(pop from stack)
s  s  u  u  b  b  T1
T1----->t  T
s  s  u  u  b  b  t  T
terminals match
s  s  u  u  b  b  t  T
T----->v  V  t  T
s  s  u  u  b  b  t  v  V  t  T
terminals match
s  s  u  u  b  b  t  v  V  t  T
V----->\L
s  s  u  u  b  b  t  v  t  T
```

```
terminals match
s  s  u  u  b  b  t  v  t  T
T----->\L  T1
s  s  u  u  b  b  t  v  t  \L  T1
terminals mismatch(pop from stack)
s  s  u  u  b  b  t  v  t  T1
T1----->\L
s  s  u  u  b  b  t  v  t
input matches the grammar with errors
```

- **Example 4:**

```
# S =  a S | b X
# X = X X c | X d | Y
# Y = Y e | f | g
```

```
First:
first of S : a b
first of a S  : a
first of b X  : b
first of X : f g
first of Y X'  : f g
first of X' : f g d \L
first of Y X' c X'  : f g
first of d X'  : d
first of \L  : \L
first of Y : f g
first of f Y'  : f
first of g Y'  : g
first of Y' : e \L
first of e Y'  : e
first of \L  : \L

Follow:
follow of S : $
follow of X : $
follow of X' : $ c
follow of Y : f g d $ c
follow of Y' : f g d $ c
```

```
Parsing table:
non-terminal: S
 on $ gives the production: synch
 on a gives the production: a S
 on b gives the production: b X
non-terminal: X
 on $ gives the production: synch
 on f gives the production: Y X'
 on g gives the production: Y X'
non-terminal: X'
 on $ gives the production: \L
 on c gives the production: \L
 on d gives the production: d X'
 on f gives the production: Y X' c X'
 on g gives the production: Y X' c X'
non-terminal: Y
 on $ gives the production: synch
 on c gives the production: synch
 on d gives the production: synch
 on f gives the production: f Y'
 on g gives the production: g Y'
non-terminal: Y'
 on $ gives the production: \L
 on c gives the production: \L
 on d gives the production: \L
 on e gives the production: e Y'
 on f gives the production: \L
 on g gives the production: \L
```

For the input :

```
a b g e f c
```

Tokens of lexical :

```
Lexical Output:
token: a value: a
token: b value: b
token: g value: g
token: e value: e
token: f value: f
token: c value: c
```

The stack :

```
S
S----->a  S
a  S
terminals match
a  S
S----->b  X
a  b  X
terminals match
a  b  X
X----->Y  X'
a  b  Y  X'
Y----->g  Y'
a  b  g  Y'  X'
terminals match
a  b  g  Y'  X'
Y'----->e  Y'
a  b  g  e  Y'  X'
terminals match
a  b  g  e  Y'  X'
Y'----->\L
a  b  g  e  X'
X'----->Y  X'  c  X'
a  b  g  e  Y  X'  c  X'
Y----->f  Y'
a  b  g  e  f  Y'  X'  c  X'
terminals match
a  b  g  e  f  Y'  X'  c  X'
Y'----->\L
a  b  g  e  f  X'  c  X'
X'----->\L
a  b  g  e  f  c  X'
terminals match
a  b  g  e  f  c  X'
X'----->\L
a  b  g  e  f  c
input matches the grammar
```

Function of phases(1 & 2) :

Phase 1:

The function of this phase is to generate a lexical analyzer given a set of regular expressions and definitions. Then using this lexical analyzer we generate the tokens corresponding to an input file and pass them to the next phase.

In the main function we start by reading the input example from the input file and pass it to the lexical analyzer through the function get_next_token() and store all these tokens in a vector of strings.

Phase 2:

The function of this phase is to produce a syntax analyzer given a grammar. It performs left factoring and left recursion elimination on the productions and then generates the first and follow for each non terminal. It then produces the parsing table to perform the syntax analysis on a token stream obtained from the previous phase.

We directly call this vector of string added to it ('$') to be our string that is compared with the terminals or non terminals of the stack to check whether the input matches the grammar or not.

ASSUMPTIONS :

- In the input file productions, the terminals and non terminals are separated by spaces as no other separation method was stated. Also, non terminals are separated by the '#' character as the first character in the line containing a new non terminal so that the productions are multilined. These assumptions are justified by the given information and input file example.
- This is an **important** assumption for the input file:
    1. For test1 the ',' token isn't defined in the given grammar so it gives an error during syntax analysis.
    - For test2 the '=' operator is defined in the lexical definition as 'assign op' while in the syntax grammar definition as '=', so we had to change it in the syntax grammar file to 'assign op' to be matched with its corresponding token obtained from the lexical analyzer.
- The epsilon terminal is defined as the string of single character "\\L" justification is that it is required to print it in the form of "\L" to represent epsilon and the backslash is an escape character.
- The terminating symbol is "$" and the synchronizing token is "synch" justification is to have the same convention as that studied in the lecture.
- Dealing with the error tokens in the syntax analysis stage is by neglecting them, justification is that we need to handle their errors in the lexical phase as is already done while in this phase

we handle syntactic errors. In case of presence of error token obtained from the first phase, the error tokens are removed. (They are already shown in the lexical analysis output).

- Before starting the syntax analysis operation on the given input, if the grammar isn't LL(1) (ie: double entry in the parsing table), syntax analysis isn't done, instead an appropriate error message is printed. Justification is that in case of double entry in the parsing table the syntax analysis won't work correctly any way due to ambiguity of the grammar.
- We get all tokens of the input from the lexical analysis phase at once instead of calling get next token each time. Justification is that this facilitates processing errors in the tokens and manipulating the tokens in syntax analysis as a given array of tokens.
- This assumption is concerning the bonus part for eliminating left recursion. We assumed that no input production would be in the form of A -> A a1| A a2| A a3 as this would cause one of two cases:
    1. The need to add the production A-> \L as eliminating left recursion gives A->A' and A'-> a1 A'|a2 A'.....|\L and has the derivation A-> A' -> \L which can't be achieved by A->A a1|A a2...
    - If we don't add this production, we would get an infinite recursion anyways as for example: A->Aa gives A->Aaa->Aaaa->Aaaaa..... Without ending.

Phase III

Evaluation of Semantic Actions in Predictive Non-Recursive Parsing paper summary

This paper we used to evaluate the semantic actions of LL1 grammar

Abstract

To implement a syntax-directed translator, compiler designers always have the option of building a compiler that first performs a syntax analysis and then transverses the parse tree to execute the semantic actions in order. Yet it is much more efficient to perform both processes simultaneously. This avoids having to first explicitly build and afterward transverse the parse tree, which is a time- and resource-consuming and complex process.
This paper introduces an algorithm for executing semantic actions (for semantic analysis and intermediate code generation) during predictive non-recursive LL(1) parsing. The proposed method is a simple, efficient, and effective method for executing this type of parser and the corresponding semantic actions jointly with the aid of no more than an auxiliary stack.

Translation schema

A translation scheme is a context-free grammar in which attributes are associated with the grammar symbols and semantic actions are inserted within the right sides of productions. These semantic actions are enclosed between brackets { }. The attributes in each production are computed from the values of the attributes of grammar symbols involved in that production. There are two kinds of attributes: synthesized and inherited. An attribute is synthesized if its value in a tree node depends exclusively on the attribute values of the child nodes. In any other case, it is an inherited attribute.

PROPOSED TOP-DOWN TRANSLATOR

we introduce the design of the proposed top-down translator that can output the translation (the intermediate or object code in the case of a compiler) at the same time as it does predictive non-recursive parsing. This saves having to explicitly build the annotated parse tree and then transverse it to evaluate the semantic actions.
To simplify translator design, we consider the following criteria:

- Criterion 1. A semantic action computing an inherited symbol attribute will be placed straight in front of that symbol.
- Criterion 2. An action computing a synthesized attribute of a symbol will be placed at the end of the right side of the production for that symbol.

To generate the proposed top-down translator the LL(1) parser is modified as follows. First, stack P is modified to contain not only grammar symbols but also semantic actions.
Second, a new stack (Aux) is added. This stack will temporarily store the symbols removed from stack P. Both stacks are extended to store the attribute values (semantic information).
There is a pointer to the top of each stack. After executing the semantic action {i}, there will be another pointer to the new top (ntop) of stack P.
Let's consider the following example: X ->{1} Y1 {2} Y2... {k} Yk {k+1}, semantic action {i} will be at the top of stack P. This means that this semantic action should be executed.
We have three cases for any action:

- Case 1. The semantic action {i} computes the inherited attribute of Yi. The symbol Yi will be in stack P, right underneath action {i}, which is being executed. Thus, Yi will be the new top (ntop) of stack P at the end of this execution. The reference to an inherited attribute of Yi can be viewed as an access to stack P and, specifically, position P [ntop].
- Case 2. The semantic action {i} contains a reference to an attribute of X. As X will have already been removed from stack P when the rule X -> α was applied, the symbol X will have been entered in stack Aux. All the grammar symbols Y1, Y2... Yi-1 (preceding the semantic action {i}) will be on top of X. These symbols will have been removed from P and inserted into Aux.

- The semantic action {i} contains a reference to an attribute of some symbol of α. By definition of the L-attributed translation scheme, this attribute will necessarily belong to a symbol positioned to the left of action {i}, i.e. to one of the symbols Y1, Y2... Yi-1. These symbols will have already been moved from stack P to stack Aux. Then any reference to an attribute of any of these symbols of α can be viewed as an access to stack Aux taking into account that Yi-1 will be on Aux [top], Yi-2 will be on Aux [top – 1]... Y1 will be on Aux [top – i + 2].

TOP-DOWN TRANSLATOR ALGORITHM

the proposed top-down translator algorithm is described as follows:

- Input: An input string ω, a parsing table M for grammar G and a translation scheme for this grammar.

- Output: If ω is in L(G), the result of executing the translation scheme (translation to intermediate or object code); otherwise, an error indication.

- Method: The process for producing the translation is:
    1. Each reference in a semantic action to an attribute is changed to a reference to a position in the stack (P or Aux) containing the value of this attribute. Then the translation scheme is extended by adding a new semantic action at the end of each production. This action pops as many elements from the stack Aux as grammar symbols there are on the right side of the production.

    2. Initially, the configuration of the translator is:
       • $S is in stack P, with S (the start symbol of G) on top,
       • the stack Aux is empty, and
       • ω$ is in the input, with ip pointing to its first symbol.

    3. Repeat
       Let X be the symbol on top of stack P
       Let a be the input symbol that ip points to
       If X is a terminal Then
            If X = a Then
                 Pop X and its attributes out of stack P
                 Push X and its attributes onto stack Aux
                 Advance ip
            Else Syntax-Error ()
       If X is a non-terminal Then
            If M [X, a] = X -> {1} Y1 {2} Y2… {k} Yk {k+1}
                 Pop X and its attributes out of stack P
                 Push X and its attributes onto stack Aux
                 Push {k+1}, Yk, {k}… Y2, {2}, Y1, {1} onto stack
                 P, with {1} on top
            Else Syntax-Error ()
       If X is a semantic action {i} Then
            Execute {i}
            Pop {i} out of stack P
       Until X = $ and Aux = S

Data structures:

Class(stack_node): this node class is used to hold symbols along with their attributes and semantic actions. the modified checker stack holds these nodes.

the fields of the class are:

- **string name**: a string hold symbol label or semantic actions between brackets { }.
- **map<string, string> allAttr**: a map holds each attribute with its value.

methods of class:

- **string get_node_name():** getter for symbol name.
- **string getAttr(string attrName):** used to return the value of attribute defined as attrName if it is existed and empty string otherwise.
- **void setAttr(string attrName, string attVal):** used to set the value of attribute defined as attrName.

stack<stack_node*> checker_stack: it is a modified version of checker_stack of phase 2 in which it holds node objects of symbols and semantic actions instead of strings of symbols only . It is used during the top-down translator algorithm.

stack<stack_node*> stack_aux: a new auxiliary stack used to hold the nodes popped out of checker_stack. we need it to calculate to have accesses on these nodes to evaluate inherited attributes of symbols

map<string,pair<int,string>> symbol_table: a map implementation of the symbol table. Maps each identifier token lexeme (as X) to a pair. This pair has an integer for the value of the identifier which is an incremental number showing the location of this identifier in the local variables array, set in the lexical analysis part and a string for the type of the identifier that is set later in the semantic actions.

Functions and algorithms:

our work mainly depends on two stacks (main stack and auxiliary stack). The main stack holds terminals , non terminals and actions of productions according to the parsed input. The auxiliary stack holds only terminals and nonterminals that popped from the main stack.

We check for every element that popped from the main stack. if it matches the format of action that popped from the main stack then it is directly executed by passing the rule to our parsing algorithm.

Parse semantic(string semantic action):

each semantic action is string that takes the format {action1 # action2 # action3 # ....} then we need to split the semantic action by the delimiter ('#') and then each action is passed to function (final parser).

final parser(string action):

each action is executed in this function by one of the following four possibilities which are :

- setting attributes by calling a specific node in one of two stacks and set a new attribute of this node by the value in the right hand side.
- incrementing address according to instruction length in java bytecode.
- popping nodes from the auxiliary stack by a value that is equal to the number of terminals and nonterminals of the production that contains this action.
- printing the java bytecode that corresponds to the input. the code is concatenated in the attribute (code) of non terminals (that are specified according to the input) and finally it is stored in the attribute (code) of the nonterminal (method body).

set_attributes(string action):

the parameter of this function is action that matches the first case in the **final parser** function possibilities in which it takes the form of (attribute1 = attribute2 + newline + attribute3 + ...) then we separate the action by the delimiter ('=') to (left hand side(LHS) and right hand side(RHS)) and RHS is passed to function (**get_all_RHSelements**) which has one parameter which is vector of string that takes every string of RHS after splitting it by delimiter ('+') and evaluate this string and concatenate it to the next one and so on . Finally we set the attribute of LHS by the value of RHS.

run_stack():

it is the main function we used to check if the grammar matches the input java program. it has the same logic as phase 2 but we had to add some modifications in which :

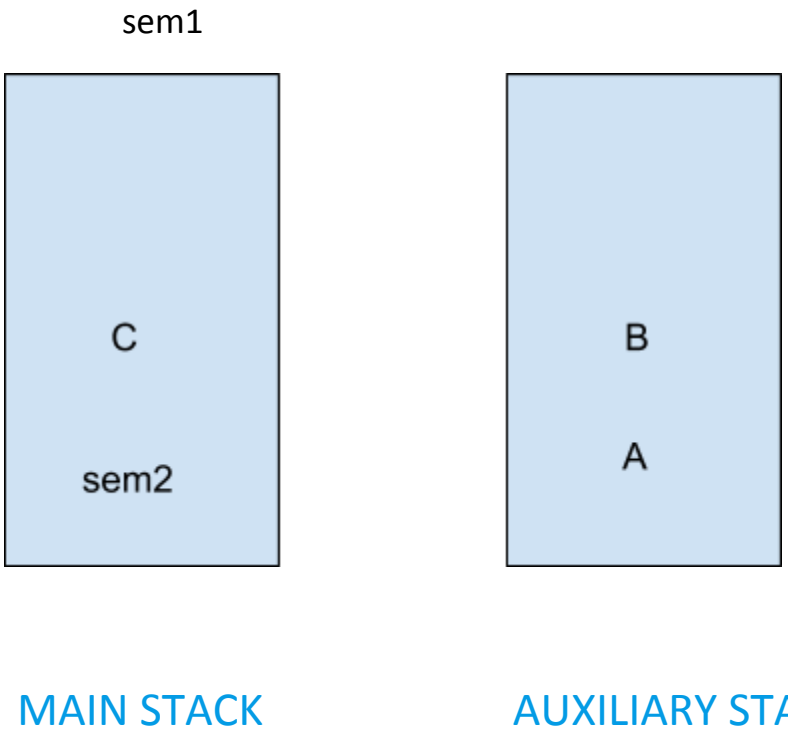
- we deal with two stacks (instead of one) each one holds nodes (**stack_node**) (instead of strings). we need nodes to hold the name and attributes of each nonterminal.
- we need to check for three cases not two because the main stack now holds (terminal , nonterminal and actions) . if the top of stack matches the format of a semantic action then we pass this action to function (**parse semantic**) to start its execution.
- every terminal or nonterminal popped from the main stack is pushed to the auxiliary stack that is used for setting synthesized attributes or for evaluating RHS of inherited ones.

For each action parsed , the terminals or nonterminals that we refer to on RHS of the action are in auxiliary stack and on LHS are in the main stack

eg: let the first semantic action after (B) is (sem1) and the first one after (C) is (sem2) and we currently have the top of main stack is (sem1)

A ----> B { action1 # action2 } C { action1 # action2 # pop(2)}

we need to change each reference to a terminal or nonterminal to its position in a specific stack.



so if one of actions in (sem1) refer to nonterminal (A) in auxiliary stack then this attribute will be AUX[1].val (instead of A.val) and the value of (1) indicates the second element of the stack and if it refer to element c in main stack then the attribute will be S[0].val.

For sem2 the action will be adjusted to include one more action(pop(2)) which pop non terminals (B) and (C) from the auxiliary stack because we propagate their values of attributes to the head non terminal (A) so need for them in the auxiliary stack

Backpatching

Backpatching is a process in which the operand field of an instruction containing a forward reference is left blank initially. The address of the forward reference symbol is put into this field when its definition is encountered in the program.

Why do we need it ?

There is a problem that occurs when generating code for flow-of-control (If and While) statements is that of matching a jump instruction with the target of the jump.

Solution

A proposed solution is that we can create labels as inherited attributes to where the relevant jump instructions were generated.But a second pass is then needed to bind labels to addresses.

Another solution is using backpatching, in which lists of jumps, containing the addresses that we need to jump form, are passed as synthesized attributes.

When a jump is generated, the targeted label (the label which we need to jump to) is left unspecified at first.Each such jump is added to a list of jumps whose labels will be filled in when the proper label can be determined.All of the jumps on a list have the same target label, basically all the addresses in the list need to jump to the same address.

Needed definitions

These definitions is proposed by our reference book Compilers Principles Techniques and Tools:

B.truelist will be a list of jump or conditional jump instructions into which we must insert the label to which control goes if a boolean expression B is true.

B.falselist will be a list of instructions that get the label to which control goes when a boolean expression B is false.

S.nextlist is a list of jumps to the instruction immediately following the code for statement S.

We generate instructions into an instruction array, and labels will be indices into this array.To manipulate lists of jumps, we use three functions: i-index to array of instructions, p-pointer to a list

1. ***makelist(i):*** creates a new list containing only i and returns a pointer to the created list.
2. ***merge(pl , p2):*** concatenates the lists pointed to by pl and p2, and returns a pointer to the

concatenated list.
3. ***backpatch(p, i):*** inserts i as the target label for each of the instructions on the list pointed to by p.

Dealing with jump statements:

we need to add some modifications to actions above to deal with if and while statements :

- we need to add actions directly after each statement and expression to increment the address correctly after calculating the code of each expression and statement.

- we marked the places where we need to jump from by labels that increments directly after writing them (L1,L2,L3,.....).

- we store the target address jump in a vector and then we loop on each instruction in the code and match it by its corresponding target address.

- we arrange the addresses to match the correct start and ending of jump statements.

Semantic actions used

JAVA byte code:

It is the low level representation of the JAVA language code that can be recognized by the machine after being converted by an assembler to binary. It is present in the class file generated when compiling JAVA. It can be assembled by a JAVA assembler as JASMIN converting it to binary. Consists of directives as .class, labels and instruction set consisting of 255 instructions. These instructions depend on a stack based architecture where operands are pushed onto the stack and result of operation is at top of stack after executing it. Each instruction has a certain number of bytes needed to represent it, so with each instruction the address of the start of this instruction is indicated. The used instructions in our case are:

- iconst/fconst: size 1 byte, loads a constant operand to stack top.
- iadd/fadd/isub/fsub/imul/fmul/idiv/fdiv: size 1 byte, performs the operation between the two top elements on the stack.
- istore/fstore: size 1 byte, stores stack top to the variable identified by its value (position in local variable array).
- iload/fload: size 1 byte, loads the variable identified by its value onto top of the stack.
- icmpl/fcmpl: size 1 byte, compares the two values on the stack top placing 1 in case of greater 0 in case of equal and -1 in case of smaller onto the stack top.
- if ge/if le/if eq/if ne/if lt/if gt: size 1 byte, compares the value on stack top to 0 and jumps to a certain address accordingly.
- goto: size 3 bytes, jumps unconditionally to a certain address.

JVM:

Virtual processor on which JAVA runs, interfaces between JAVA program and hardware. It has stack based architecture, for each method a frame is created on the stack having an array of local variables for reference to parameters and local variables. Zero slot in the local array variable is reserved for **this** in case of instance methods. When a variable or a constant is called, its value is pushed to the stack.

Implemented data types:

- Int.
- Float.

Supported JAVA code:

- Declaration: We just change the type of the identifier accordingly.
- Assignment: The result of the expression is calculated then stored in the variable by using istore/fstore instruction.
- Arithmetic expression: Calculated by placing each two operands on stack and then calling the correct instruction according to the operation to be performed on them.

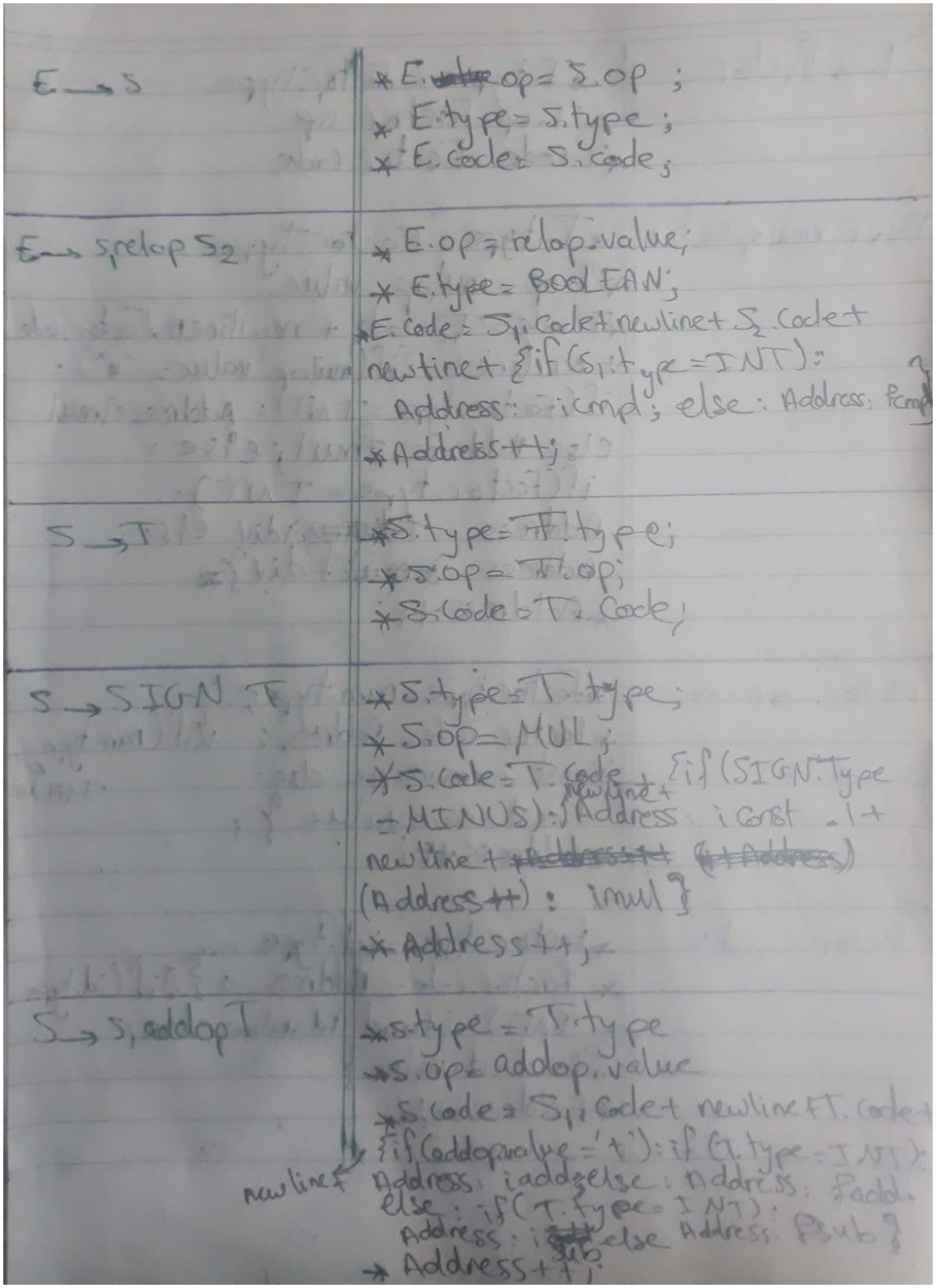
- If: According to the calculated expression, the type of conditional jump is identified (which is opposite to the comparison operation performed in the expression) and correct instruction is placed.
- If - else: Same as case of if but with the addition of a goto statement before the statement after the else clause to jump out of the else clause in case the if condition was true.
- While: Same as if but with goto statement at end of while statement in order to loop back to the start of the while statement.

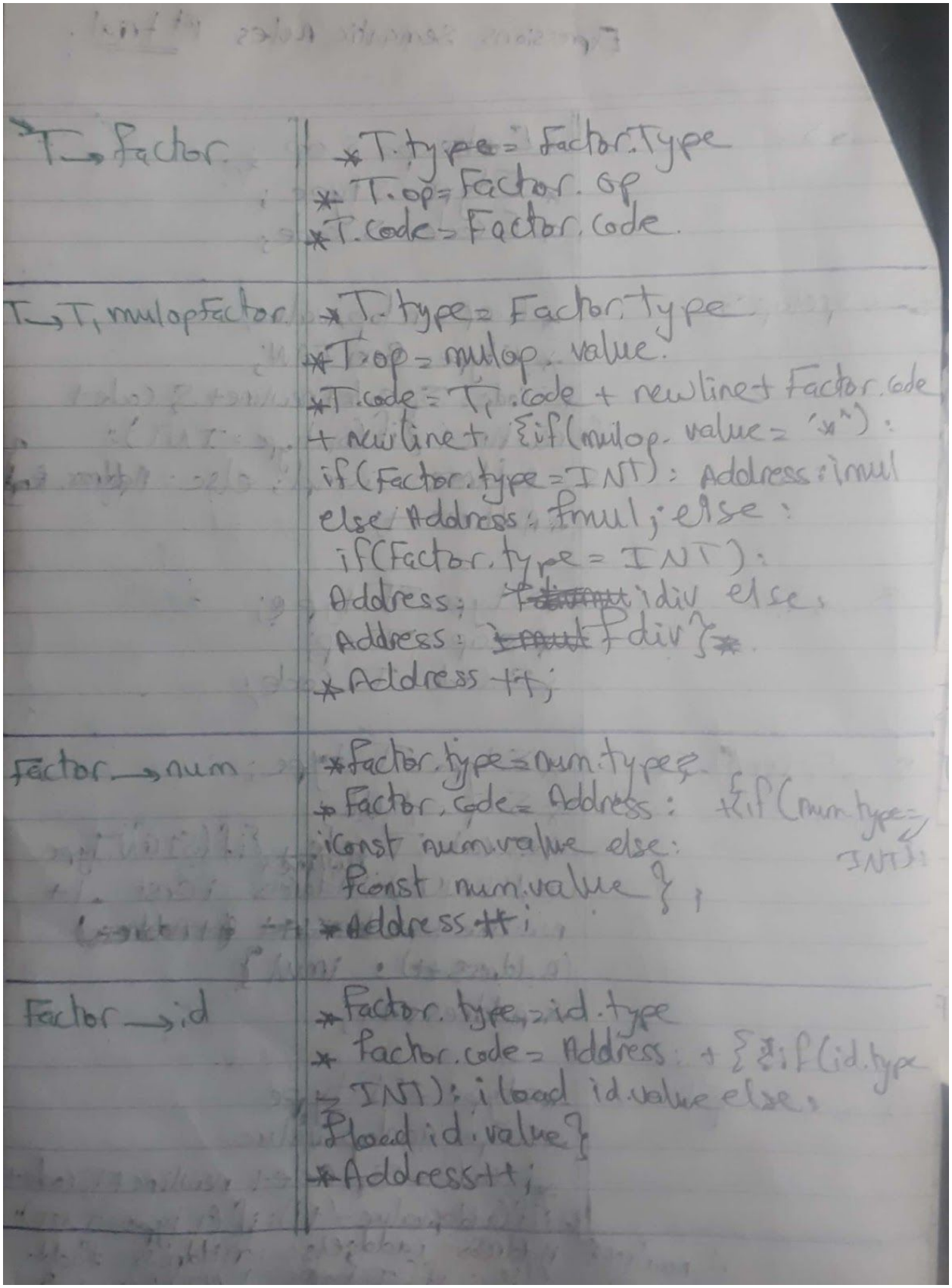
View on semantic rules:

- The main idea is to use the synthesized and inherited attributes in order to obtain a final code for the byte-code translation at the end to be printed. The variables are mainly to hold the operation type of the expression (to identify the type of conditional jump in case of boolean expressions), code to be stored in this nonterminal (to collect the byte-code step by step till the end) and type to store the type of the operation for type checking(if an extension was to be needed).
- Used attributes from terminals are:
    - value which refers to the numerical value of a number and the location in the local variable array of an identifier.
    - type: to refer to the type of identifier or number (int or float).
- The rules also keep track of the address of the instruction and increments iit accordingly.
- These rules directly output the byte code (no other intermediate representation was needed).
- The studied approach in the book was used to convert the rules to those corresponding to the productions after left factoring and elimination of left recursion.
- The rules were directly inserted into the productions after processing to remove left recursion and perform left factoring to be used in performing the semantic actions.

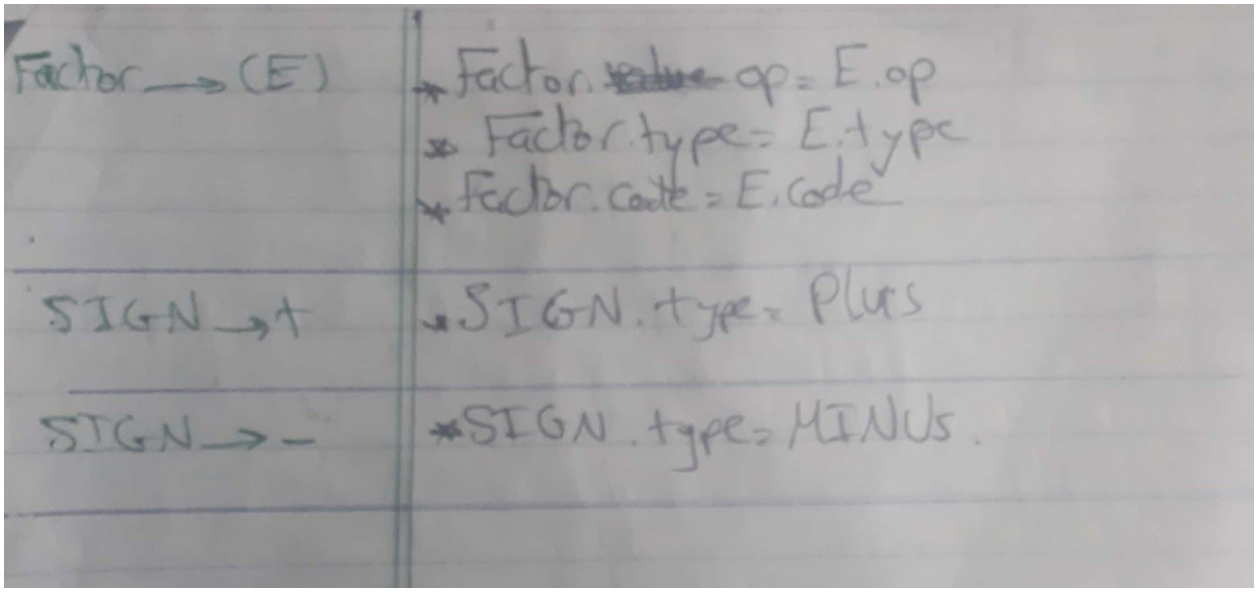
Following is part of the semantic rules for implementation of declaration, assignment and expressions byte-code.

Semantic rules before performing left recursion elimination and left factoring

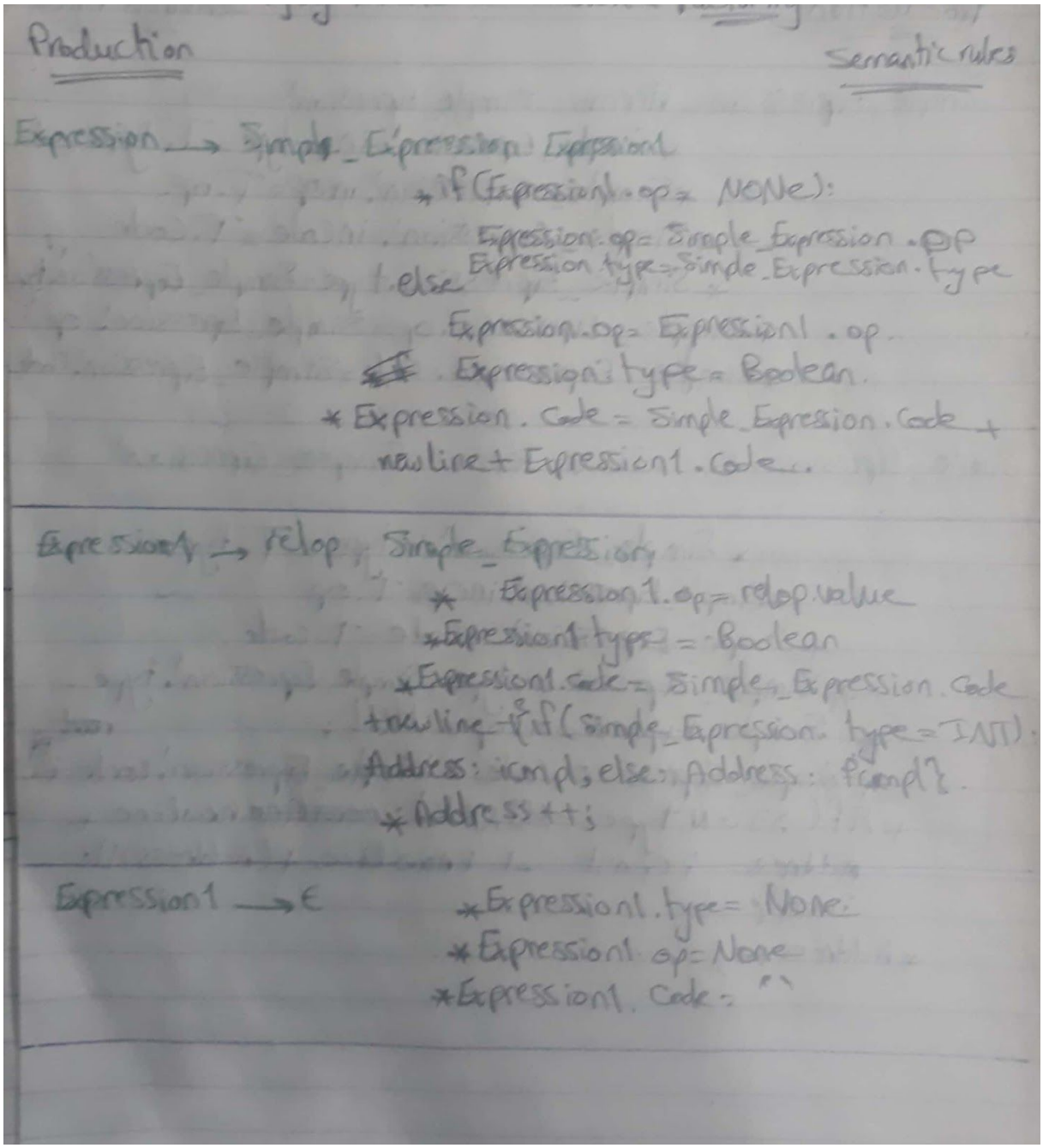

E → S
* E.op = S.op ;
* E.type = S.type ;
* E.code = S.code ;
E → S1 relop S2
* E.op = relop.value ;
* E.type = BOOLEAN ;
* E.code = S1.code + newline + S2.code + newline + {if (S1.type = INT): Address: icmp ; else : Address: fcmp}
* Address++ ;
S → T
* S.type = T.type ;
* S.op = T.op ;
* S.code = T.code ;
S → SIGN T
* S.type = T.type ;
* S.op = MUL ;
* S.code = T.code + newline + {if (SIGN.type = MINUS): Address: iconst -1 + newline + (Address++) : imul}
* Address++ ;
S → S1 addop T
* S.type = T.type
* S.op = addop.value
* S.code = S1.code + newline + T.code + newline + {if (addop.value = '+'): if (T.type = INT): Address: iadd else : Address: fadd. else : if (T.type = INT): Address: isub else Address: fsub}
* Address++ ;

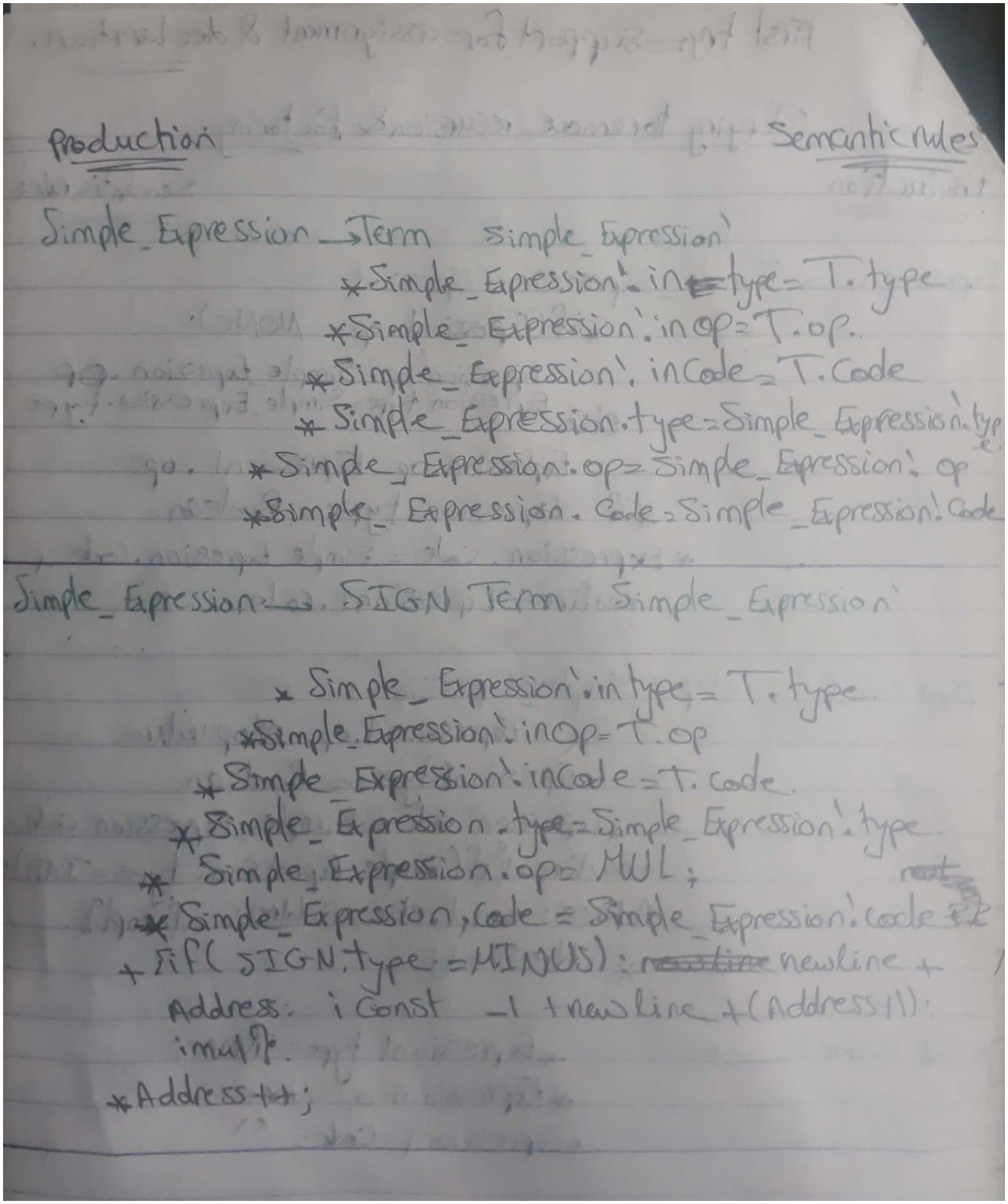


T → factor	*T.type = factor.type *T.op = Factor.op *T.code = Factor.code
T → T1 mulop factor	*T.type = Factor.type *T.op = mulop.value *T.code = T1.code + newline + Factor.code + newline + {if(mulop.value = '*'): if(Factor.type = INT): Address: imul else Address: fmul; else: if(Factor.type = INT): Address: idiv else: Address: fdiv}* *Address ++;
factor → num	*factor.type = num.type; *Factor.code = Address: {if(num.type = INT): iconst num.value else: fconst num.value}; *Address ++;
Factor → id	*Factor.type = id.type * Factor.code = Address: {if(id.type = INT): iload id.value else: fload id.value} *Address++;


Factor → (E)
Factor.op = E.op
Factor.type = E.type
Factor.code = E.code
SIGN → +
SIGN.type = Plus
SIGN → -
SIGN.type = MINUS

Semantic rules after performing left recursion elimination and left factoring





Production Semantic rules

Simple_Expression → Term Simple_Expression'

*Simple_Expression'.inType = T.type

*Simple_Expression'.inOp = T.op

*Simple_Expression'.inCode = T.Code

*Simple_Expression.type = Simple_Expression'.type

*Simple_Expression.op = Simple_Expression'.op

*Simple_Expression.Code = Simple_Expression'.Code

Simple_Expression → SIGN Term Simple_Expression'

*Simple_Expression'.inType = T.type

*Simple_Expression'.inOp = T.op

*Simple_Expression'.inCode = T.code

*Simple_Expression.type = Simple_Expression'.type

*Simple_Expression.op = MUL;

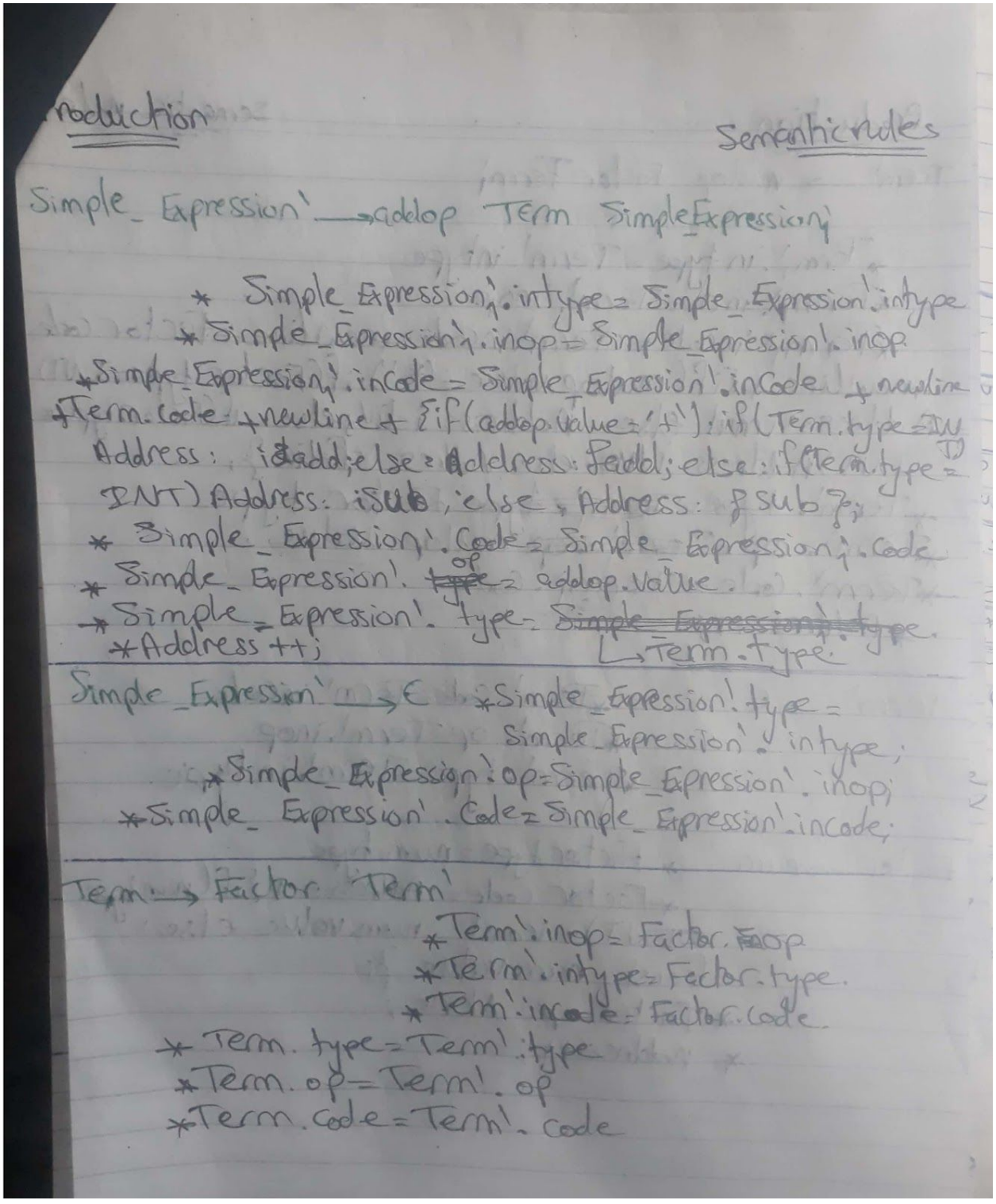
*Simple_Expression.code = Simple_Expression'.code

+if(SIGN.type = MINUS): newline +

Address: iConst -1 + newline + (Address+1):

imul

*Address++;



roduction

Semantic rules

Simple_Expression' → addop Term Simple_Expression'

* Simple_Expression'1.intype = Simple_Expression'.intype
* Simple_Expression'1.inop = Simple_Expression'.inop
* Simple_Expression'1.incode = Simple_Expression'.incode + newline + Term.code + newline + {if(addop.value='+'); if(Term.type=INT) Address: iadd; else Address: fadd; else: if(Term.type=INT) Address: isub; else Address: fsub;}
* Simple_Expression'.Code = Simple_Expression'1.code
* Simple_Expression'.~~type~~ op = addop.value.
* Simple_Expression'.type = ~~Simple_Expression'1.type~~ → Term.type
* Address ++;

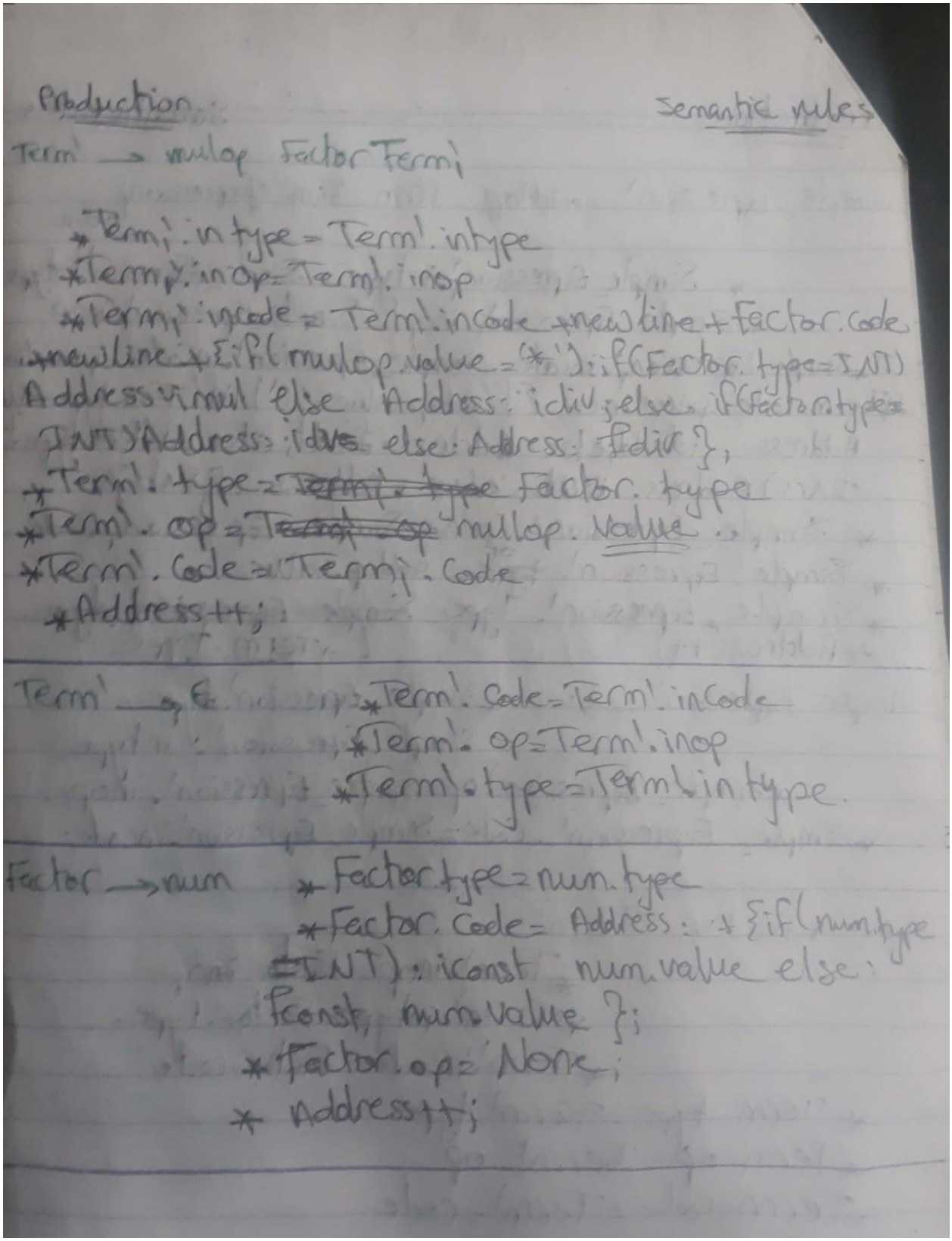
Simple_Expression' → ε

* Simple_Expression'.type = Simple_Expression'.intype;
* Simple_Expression'.op = Simple_Expression'.inop;
* Simple_Expression'.Code = Simple_Expression'.incode;

Term → Factor Term'

* Term'.inop = Factor.~~F~~op
* Term'.intype = Factor.type.
* Term'.incode = Factor.code

* Term.type = Term'.type
* Term.op = Term'.op
* Term.code = Term'.code



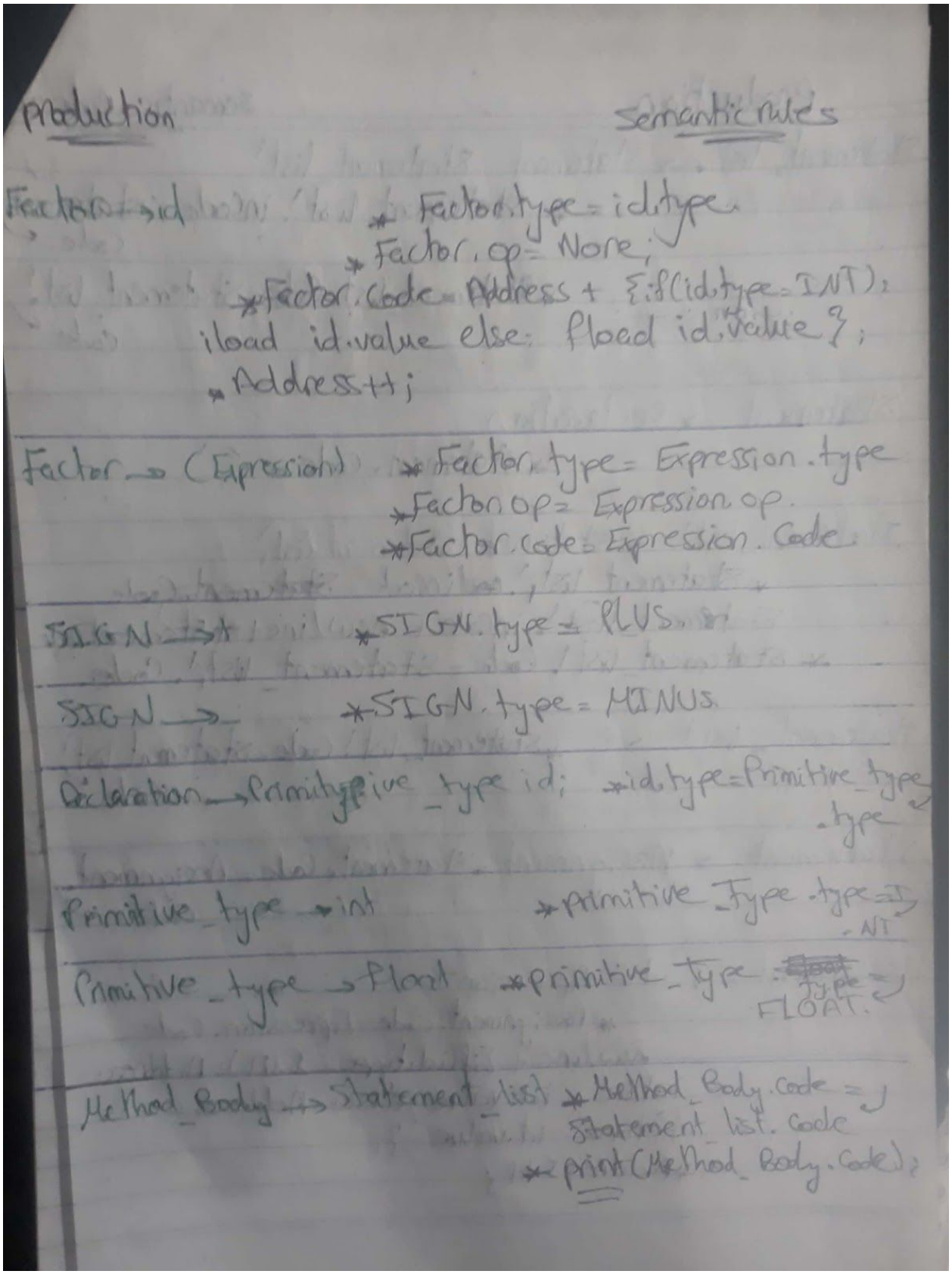
Production | Semantic rules

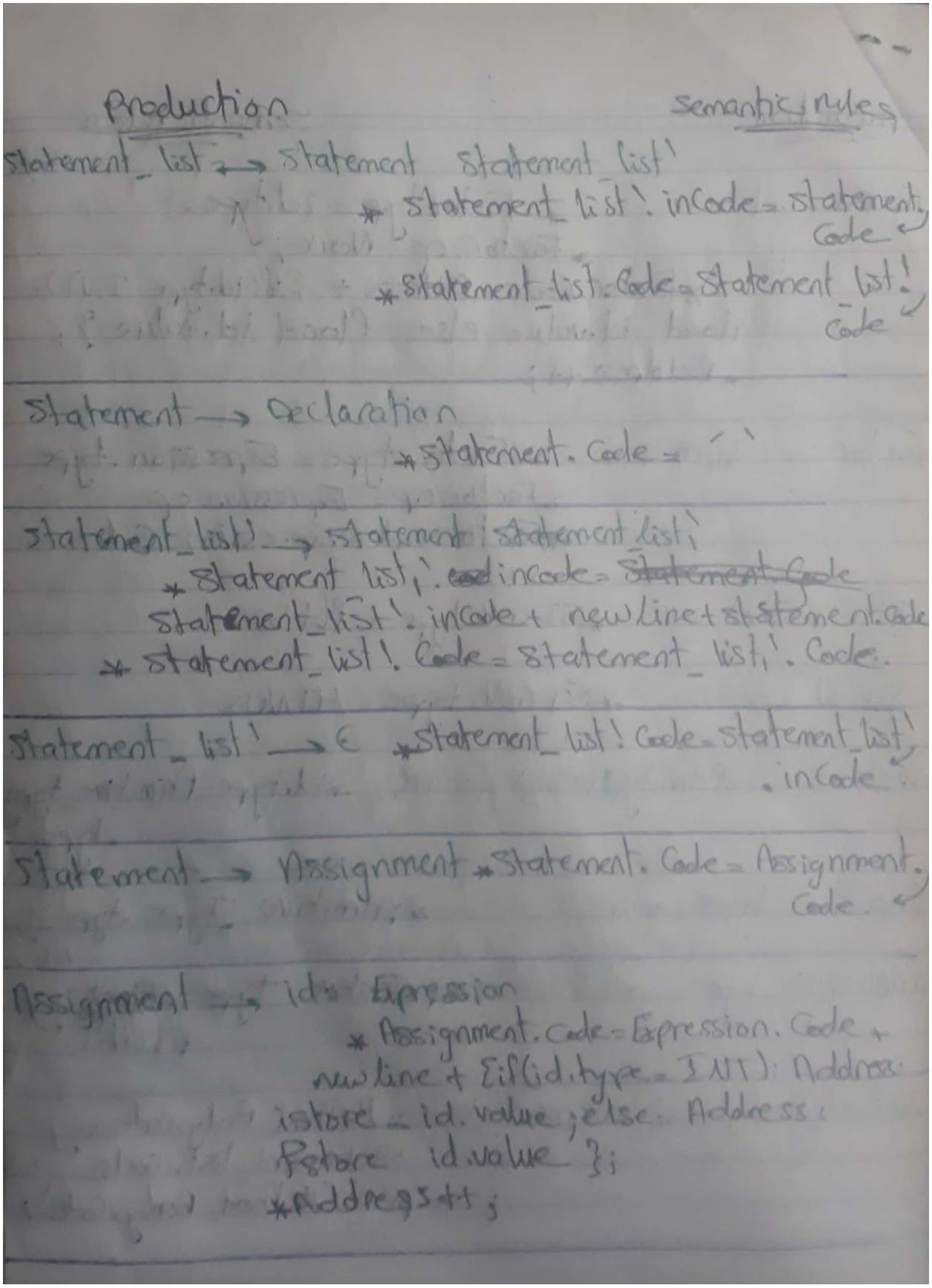
Term' → mulop Factor Term1

*Term1.intype = Term'.intype
*Term1.inop = Term'.inop
Term1.incode = Term'.incode + newline + Factor.code + newline + {if(mulop.value = ''); if(Factor.type = INT) Address: imul else Address: idiv; else if(Factor.type = INT) Address: idvs else Address: fdiv},
*Term'.type = ~~Term'.type~~ Factor.type
*Term'.op = ~~Term'.op~~ mulop.value
*Term'.code = Term1.code
*Address++;

Term' → ε
*Term'.code = Term'.inCode
*Term'.op = Term'.inop
*Term'.type = Term'.intype

Factor → num
*Factor.type = num.type
*Factor.code = Address: + {if(num.type = INT): iconst num.value else: fconst, num.value};
*Factor.op = None;
*Address++;


Production
Semantic rules
Factor → id
* Factor.type = id.type
* Factor.op = None;
* Factor.code = Address + {if(id.type = INT): iload id.value else: fload id.value};
* Address++;
Factor → (Expression)
* Factor.type = Expression.type
* Factor.op = Expression.op.
* Factor.code = Expression.Code
SIGN → +
* SIGN.type = PLUS
SIGN → -
* SIGN.type = MINUS
Declaration → Primitive_type id;
* id.type = Primitive_type .type
Primitive_type → int
* Primitive_Type.type = INT
Primitive_type → float
* primitive_type.type = FLOAT.
Method_Body → Statement_list
* Method_Body.code = Statement_list.code
* print(Method_Body.code);


Production
Semantic rules
Statement_list → Statement Statement_list'
* Statement_list'.inCode = Statement.Code
* Statement_list.Code = Statement_list'.Code
Statement → Declaration
* Statement.Code = ' '
Statement_list' → Statement Statement_list'
* Statement_list'.inCode = Statement.Code
Statement_list'.inCode + newline + statement.Code
* Statement_list'.Code = Statement_list'.Code.
Statement_list' → ∈ * Statement_list'.Code = Statement_list'.inCode
Statement → Assignment * Statement.Code = Assignment.Code
Assignment → id = Expression
* Assignment.Code = Expression.Code +
newline + {if(id.type = INT) Address
istore = id.value; else Address
fstore id.value };
*Address++;

Semantic rules for control-flow statement using backpatching

This is a trial, the actions are not 100% correct.

Production Rules	Semantic Rules
STATEMENT -> IF	|STATEMENT.code = IF.code |STATEMENT.next = IF.next
STATEMENT -> WHILE	|STATEMENT.code = WHILE.code |STATEMENT.next = WHILE.next
IF -> if (EXPRESSION) MARKER1 { STATEMENT } MARKER else MARKER2 { STATEMENT }	|backpatch(EXPRESSION.true, MARKER1.instr) |emit('if' EXPRESSION 'goto' MARKER1.instr) #true# |backpatch(EXPRESSION.false, MARKER2.instr) |emit('goto' MARKER2.instr) #false# |IF.next = merge(STATEMENT.next, MARKER.next)
WHILE -> while MARKER1 (EXPRESSION) MARKER2 { STATEMENT }	|backpatch(EXPRESSION.true, MARKER2.instr) |emit('if' EXPRESSION 'goto' MARKER2.instr) #true# |emit('goto') #false-end of loop# |backpatch(STATEMENT.next, MARKER1.instr) |WHILE.next = EXPRESSION.false |emit('goto' MARKER1.instr) #repeat#
MARKER -> \L	|MARKER.next = makelist(nextinstr) |emit('goto') #next statement#
MARKER1 -> \L	|MAREKER1.instr = MARKER1.nextinstr
MAREKR2 -> \L	|MAREKER2.instr = MARKER2.nextinstr

Sample Runs :

test for while statements:

```
int x;
x=5;
while ( x>6) {x=9;}
```

```
0:iconst 5
1:istore_1
2:iload 1
3:iconst 6
4:icmpl
5:if_le  #11
6:iconst 9
7:istore_1
8:goto  #2
```

test for if else statements:

```
int x;
x = 5;
if(x < 6) {x=8; } else {x=9;}
```

```
0:iconst 5
1:istore_1
2:iload 1
3:iconst 6
4:icmpl
5:if_ge  #11
6:iconst 8
7:istore_1
8:goto  #13
11:iconst 9
12:istore_1
```

tests for nested if statements:

```
int x;
x = 5;
if(x < 6) { if(x >= 7) { x=13; } else {x=11;} } else {x=12;}
```

```
0:iconst 5
1:istore_1
2:iload 1
3:iconst 6
4:icmpl
5:if_ge  #20
6:iload 1
7:iconst 7
8:icmpl
9:if_lt  #15
10:iconst 13
11:istore_1
12:goto  #17
15:iconst 11
16:istore_1
17:goto  #22
20:iconst 12
21:istore_1
```

tests for declaration and assignment statements:

- **Test 1**

```
int x;
int y;
float z;
x = 3 + 5;
y = 6 * 9;
z = x / y;
```

```
0:iconst 3
1:iconst 5
2:iadd
3:istore_1
4:iconst 6
5:iconst 9
6:imul
7:istore_2
8:iload 1
9:iload 2
10:idiv
11:fstore_3
```

- **Test 2**

```
int x;
int y;
x = (3 + 5) * 7;
y = 5 * x + 4 ;
```

```
0:iconst 3
1:iconst 5
2:iadd
3:iconst 7
4:imul
5:istore_1
6:iconst 5
7:iload 1
8:imul
9:iconst 4
10:iadd
11:istore_2
```

- **Test 3**

```
int x;
float y;
float z;
x = 3 + (5 - 7) * 3;
y = (x / 3) * (5 + 7);
z = y / (3 * x);
```

```
0:iconst 3
1:iconst 5
2:iconst 7
3:isub
4:iconst 3
5:imul
6:iadd
7:istore_1
8:iload 1
9:iconst 3
10:idiv
11:iconst 5
12:iconst 7
13:iadd
14:imul
15:fstore_2
16:fload 2
17:iconst 3
18:iload 1
19:imul
20:idiv
21:fstore_3
```

tests for sign before numbers:

```
int x;
x = (minus 3) + 5;
```

```
0:iconst 3
1:iconst -1
2:imul
3:iconst 5
4:iadd
5:istore_1
```

Assumptions :

- For the nonterminal (SIGN) , it contains productions either (‘+’ or ‘-’) terminals but these terminals are used for regular expression (addop) so we modified the rules of phase 1 and added new keywords (plus and minus) and modified the grammar of phase 2 to make the nonterminal (SIGN) holds (‘plus’ or ‘minus’). We then removed the modification after testing for the semantic rules handling negative numbers.

Conclusion :

Our work has provided a program that generates a lexical and a syntax analyzer given the lexical rules and the syntax grammar. This gives a dynamic method for compiler generation for any well defined programming language in quite an efficient manner. Furthermore, we defined semantic rules in the syntax grammar to generate Java byte code corresponding to a subset of the Java language containing declaration, assignment and arithmetic expressions as well as if, if else and while conditions but without providing the jump addresses. We also provided an implementation to perform these rules and obtain the byte code as output in a file.

Roles:

- Hazem Morsy:
  - Phase1:
    - helping in parser
    - filling NFA with needed data from parser
  - Phase2:
    - parsing of input file.
    - building the stack to check the grammar according to the parser table and solving errors by panic mode recovery.
  - phase 3:
    - modifying some functions in phase 2 to help us in phase 3.
    - parsing and execution of semantic actions.

- Ahmed Ali:
  - Phase1:
    - Failed trial in DFA, due to an overflow problem. But the general algorithm implementation was used later as it was correct.
  - Phase2:
    - Calculation of follow set.
    - Filling the parsing table.
    - Helped in making the model, the suggestion to separate between terminals and non-terminals.
  - phase 3:
    - Backpatching semantic actions for control-flow statement (Trial).

- Youssef Fathy:
  - Phase1:
    - parsing lexical rules
  - Phase2:
    - eliminate grammar left recursion
    - perform left factoring
  - phase 3:
    - testing

- Sherif Mohamed:
    - Phase1:
        - NFA creation and operations.
        - Helped in DFA creation.
        - DFA minimization.
        - Bonus part of using FLEX.
    - Phase2:
        - Helped in making the general model for the syntax analyzer objects.
        - Calculation of the first set for non terminals.
        - Helped in filling the parsing table.
    - phase 3:
        - Added the code to fill the symbol table from lexical analysis.
        - Added the semantic actions needed to the syntax grammar provided in order to output the java byte code.

Used tools:

- The IDE used was code blocks and we are mostly familiar with it. We faced a problem due to compiling the REGEX in code blocks but after updating the used version the problem was solved.
- FLEX was used in generating lexical analyzers as a bonus part for phase 1. We had to learn a new way of writing codes to define the lexical rules in an appropriate manner for this tool to work. At first using it was difficult as there isn't a unified reference manual to use it online but after searching for some time, the correct steps to use it became clear.

References:

- Our main reference for the used algorithms and methods was: **Compilers Principles Techniques and Tools** book.
- We also depended on websites like **stack overflow** and **geeks for geeks** for help in debugging errors and implementation details in our code.
- This link was used in obtaining the flex analyzer for phase 1 bonus part **http://gnuwin32.sourceforge.net/packages/flex.html**.
- This great blog by James Bloom was referenced to understand the JAVA Byte code **https://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html**.
- This reference was used to understand more details about JVM and JAVA Bytecode instruction set **http://saksagan.ceng.metu.edu.tr/courses/ceng444/link/f3jasmintutorial.html**.
- This reference is for the different parser and scanner generator tools available **https://en.wikipedia.org/wiki/Comparison_of_parser_generators**.
- we take the algorithm of evaluating semantic actions from here Innovations and Advances in Computer Sciences and Engineering pp 491-496 author José L. Fuertes Aurora Pérez