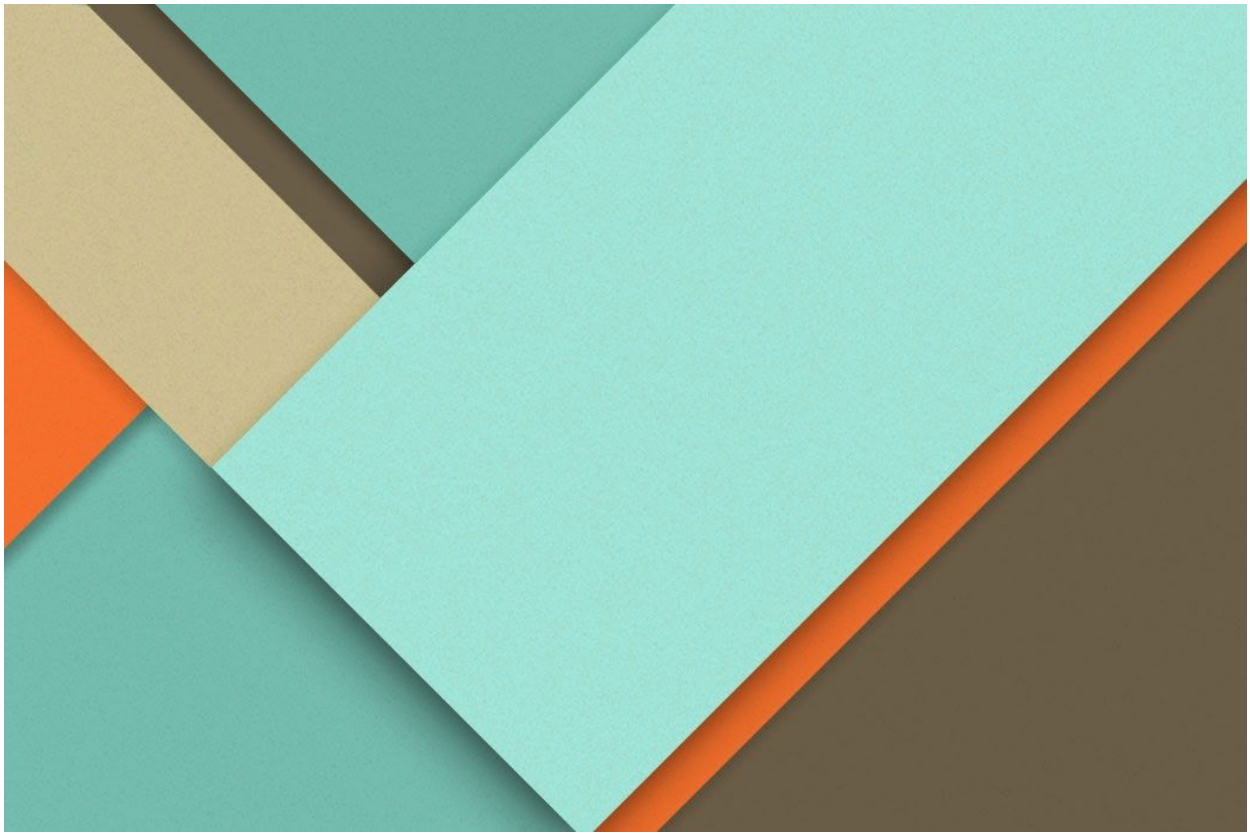


Compiler Generator

Abanoub Ashraf Ezzat Zaki ID: 1

Bassam Rageh Ibrahim ID: 15

Eman Rafik ID: 13

Reham Mohamed Naguib ID: 19

Abstract

Compiler generator is a tool used to automate the process of designing a specific compiler for a source language. It is provided with the rules defining this language and gives an auto-generated compiler. We developed two phases of a front-end compiler generator: lexical analyzer generator and parser generator with c++. Regarding the lexical analyzer generator, its input is regular expressions defining different expected tokens. It gives the transition table of minimized Deterministic Finite Automata (DFA) for the given set of rules. And as for the parser generator, it accepts context-free grammar rules for the specified language, converts it to LL(1) grammar and outputs a parsing table. A panic mode recovery technique is followed in both the generated lexical analyzer and parser. We tested the developed compiler generator phases with a minimized set of java rules and a small java program. Finally, we added suitable semantic rules to the given set of java grammar rules, the desired output of these actions is to generate java bytecode for a given program. We used GNU Bison and Flex tools to use the annotated rules and generate java bytecode.

Introduction

Splitting the compiler into a front-end and back-end with an intermediate code generation has decreased the dependence between source and target languages. Furthermore, compiler generators, also called compiler-compiler, have eliminated the need of a specific compiler for each source language. A compiler generator tool is designed to generate the different phases of a compiler tool given the rules specifying the source language. A front-end compiler generator has three phases: lexical analyzer generator, parser generator and semantic analyzer generator. The inputs for these three phases are the lexical rules, the language grammar rules and the language semantic rules, respectively. There exist well known parser generators such as: GNU Bison, JavaCC, ANTLR, etc.

JavaCC is a parser generator running under Java Runtime Environment (JRE) and generating java LL(1) top-down parser which is easy to debug, although the grammar may not be pure LL(1). The input for JavaCC is the lexical and grammar rules.

ANTLR (ANother Tool for Language Recognition) is a parser generator tool that generates lexer and parser provided the lexical and grammar rules. It generates a recursive parser in a user-specified target language. An input text to the generated parser gives an abstract syntax tree (AST) that can be compiled giving an executable.

In this project, we developed in c++ the two main parts of a front-end compiler generator: lexical analyzer generator and a parser generator. The developed program is provided with the lexical rules and grammar productions and gives a transition table for a

minimized DFA and a parsing table as outputs for the lexical analyzer generator and parser generator, respectively. Also for an input program with the provided rules, the program gives the list of tokens resulted from the lexical analysis and the parser actions. The program was tested against a minimized set of the java language rules and a java program. Furthermore, we used the GNU Bison compiler generator to generate a java bytecode for a given program. This was done through specifying semantic rules to the given set of java grammar, where we followed the standard bytecode instructions defined in Java Virtual Machine specifications. GNU Bison is a parser generator tool with grammar productions with semantic rules as an input and LALR(1) parser tables as output. Flex is used along with Bison to split the input program, that is to be parsed using the generated parser, into tokens and provide Bison with these tokens.

Problem statement

Lexical Analyzer Generator

The task in this phase of the assignment is to design and implement a lexical analyzer generator tool.

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

The generated lexical analyzer has to read its input one character at a time, until it finds the longest prefix of the input, which matches one of the given regular expressions. It should create a symbol table and insert each identifier in the table. If more than one regular expression matches some longest prefix of the input, the lexical analyzer should break the tie in favor of the regular expression listed first in the regular specifications. If a match exists, the lexical analyzer should produce the token class and the attribute value. If none of the regular expressions matches any input prefix, an error recovery routine is to be called to print an error message and to continue looking for tokens. The lexical analyzer generator is required to be tested using the given lexical rules of tokens of a small subset of Java. Use the given simple program to test the generated lexical analyzer.

Parser Generator

This phase implements an LL(1) parser generator. The parser takes a grammar as input and automatically eliminates left recursion and performs left factoring to convert it to LL(1) grammar.

Then It computes First and Follow sets and uses them to construct a predictive parsing table for the grammar.

The table is to be used to drive a predictive top-down parser. The generated parser produces the leftmost derivation for a correct input. If an error is encountered, a panic-mode error recovery routine is called to print an error message and resume parsing.

This phase is combined with the lexical analyser in phase 1.

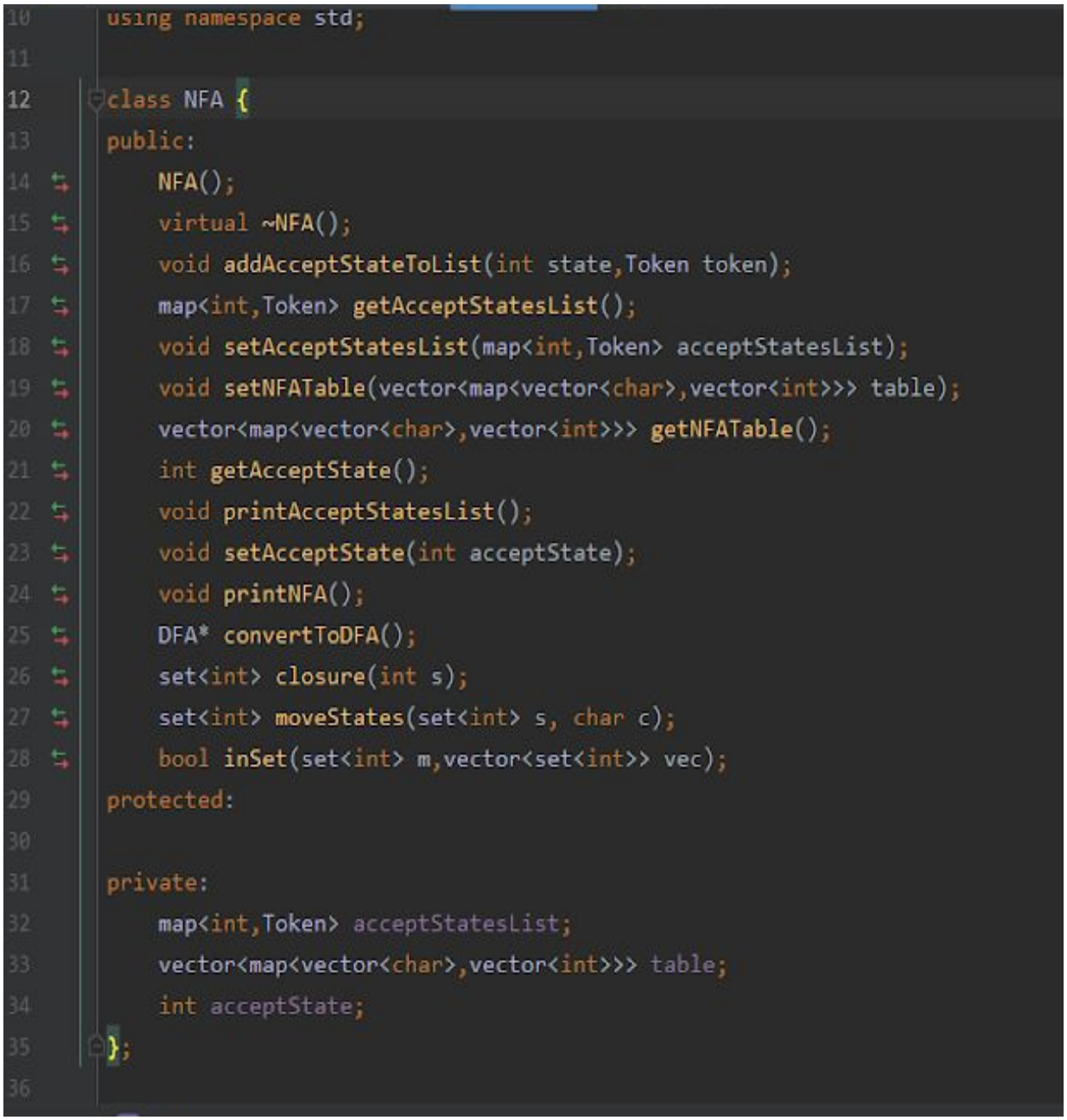
Java Byte Code Generation

This phase generates Java byte code for the input program.

Firstly we wrote semantic rules for the context free grammar to output the code. Then we implemented the parser using Bison and Lex.

Lexical Analyser Generator Design

NFA class



```
using namespace std;

class NFA {
public:
    NFA();
    virtual ~NFA();
    void addAcceptStateToList(int state,Token token);
    map<int,Token> getAcceptStatesList();
    void setAcceptStatesList(map<int,Token> acceptStatesList);
    void setNFATable(vector<map<vector<char>,vector<int>>> table);
    vector<map<vector<char>,vector<int>>> getNFATable();
    int getAcceptState();
    void printAcceptStatesList();
    void setAcceptState(int acceptState);
    void printNFA();
    DFA* convertToDFA();
    set<int> closure(int s);
    set<int> moveStates(set<int> s, char c);
    bool inSet(set<int> m,vector<set<int>> vec);
protected:

private:
    map<int,Token> acceptStatesList;
    vector<map<vector<char>,vector<int>>> table;
    int acceptState;
};

```

Attributes

- **acceptStatesList** : a list of all accept states for a particular NFA.
- **Table** : a vector that represents all the states and their transitions under the given inputs. Each state is represented by its index in the vector. The input is a vector of characters that is mapped to some states which the original state goes to under that input.
- **acceptState** : an integer that represents the accept state for a simple single accept state NFA.

Functions

- **addAcceptStateToList** : a function that takes a pair of an integer state and a token and adds them to the acceptStatesList vector.
- **getAcceptStatesList** : a function that returns the acceptStatesList vector.
- **setAcceptStatesList** : a function that sets the acceptStatesList vector of a particular NFA.
- **setNFATable** : a function that sets the NFA table.
- **getNFATable** : a function that returns the NFA table.
- **getAcceptState** : a function that returns the accept state for a single accept state NFA.
- **printAcceptStatesList** : a function to print all accept states of an NFA with many accept states.
- **setAcceptState** : a function that sets the accept state of a single accept state NFA.
- **convertToDFA :** returns a DFA corresponding to the NFA. Uses subset construction algorithm.
- **closure** : returns the epsilon closure of a given state.
- **moveStates** : returns the epsilon closure of states that a given state moves to under a given input.
- **printNFA** : a function to print the NFA table.

```
DFA* NFA::convertToDFA(){
    DFA *dfa = new DFA(0,95);
    set<int> s = closure(0);
    dfa->addState();
    vector<set<int>> vec;
    vec.push_back(s);
    int n=0;
    while(n < vec.size()){
        for(int i = 1; i < 95; i++){
            set<int> m = moveStates(vec.at(n),i+32);
            if(m.size() > 0) {
                if(!inSet(m, vec)){
                    vec.push_back(m);
                    dfa->addState();
                    dfa->addTransition(n, i,vec.size()-1);
                }else{
                    for(int j = 0; j < vec.size(); j++){
                        if(vec.at(j) == m){
                            dfa->addTransition(n, i, j);
                        }
                    }
                }
            }
        }
        n++;
    }
    map<int,Token>::iterator it = acceptStatesList.begin();
    while(it != acceptStatesList.end()){
        for(int j = 0; j < vec.size(); j++){
            set<int>::iterator setit = vec.at(j).begin();
            while(setit != vec.at(j).end()){
                if(*setit ==it->first){
                    dfa->addAcceptState(j,it->second);
                }
                setit++;
            }
        }
        it++;
    }
    return dfa;
}
```

```
set<int> NFA::closure(int st){
    set<int> res;
    stack<int> closureStack;
    closureStack.push(st);
    while(!closureStack.empty()){
        int s = closureStack.top();
        closureStack.pop();
        if(res.count(s)==0){
            res.insert(s);
            map<vector<char>,vector<int>> m = table[s];
            map<vector<char>,vector<int>>::iterator it = m.begin();
            while(it!=m.end()){
                vector<char> c = it->first;
                if(count(c.begin(), c.end(), ' ')){
                    vector<int> v = it->second;
                    for(int i=0;i<v.size();i++){
                        closureStack.push(v.at(i));
                    }
                }
                it++;
            }
        }
    }
    return res;
}

set<int> NFA::moveStates(set<int> s, char c){
    set<int> res;
    set<int>::iterator it = s.begin();
    while(it != s.end()){
        map<vector<char>,vector<int>> m = table[*it];
        map<vector<char>,vector<int>>::iterator charit = m.begin();
        while(charit!=m.end()){
            vector<char> tr = charit->first;
            int x = count(tr.begin(), tr.end(), c);
            if(x > 0){
                vector<int> v = charit->second;
                for(int i = 0; i < v.size(); i++){
                    set<int> eps = closure(v.at(i));
                    set<int>::iterator epsit = eps.begin();
                    while(epsit != eps.end()){
                        res.insert(*epsit);
                        epsit++;
                    }
                }
            }
            charit++;
        }
        it++;
    }
    return res;
}
```

DFA class

```
class DFA {
    public:
        DFA(int number_of_states, int number_of_inputs);
        DFA(int number_of_inputs);
        virtual ~DFA();
        int getNumberOfStates() const;
        int getNumberOfInputs() const;
        map<int, Token> getAcceptStates();
        int **getTable() const;
        void addTransition(int from, int input, int to);
        void addState();
        void addAcceptState(int state,Token token);
        bool isAcceptState(int state);
        bool areCompatibleStates (int state1, int state2, vector<int> partitions);
        DFA* minimize ();
        void print_dfa();

    private:
        int number_of_states, number_of_inputs;
        map<int, Token> accept_states;
        int **table;

};
```

DFA(int number_of_states, int number_of_inputs):

A constructor that creates a table of size equal to int number_of_states * int number_of_inputs, initializing it by phai which value is -1.

```
DFA::DFA(int number_of_states, int number_of_inputs) {
    this->number_of_states = number_of_states;
    this->number_of_inputs = number_of_inputs;
    //initialize table 2D array with -2s
    this->table = new int *[number_of_states];
    for (int i = 0; i < number_of_states; i++)
        this->table[i] = new int[number_of_inputs];

    for (int i = 0; i < number_of_states; i++) {
        for (int j = 0; j < number_of_inputs; j++) {
            table[i][j] = phai;
        }
    }

}
```

Void addTransition(int from, int input, int to):

Adds a transition in the dfa table from state “from” to state “to” under input “input”

Void addState():

Creates a new row in the dfa table initializing it by phai (-1).

```
void DFA::addTransition(int from, int input, int to) {
    DFA::table[from][input] = to;
}

void DFA::addState() {
    number_of_states++;
    int **new_table = new int *[number_of_states];

    for (int i = 0; i < number_of_states; i++)
        new_table[i] = new int[number_of_inputs];

    for (int i = 0; i < number_of_states; i++) {
        for (int j = 0; j < number_of_inputs; j++) {
            if (i < number_of_states - 1) {
                new_table[i][j] = table[i][j];
            } else {
                new_table[i][j] = phai;
            }
        }
    }
    this->table = new_table;
}
```

Void addAcceptState(int state, Token token):

Defines a state to be an accepted state by giving it a token instance of the accepted token.

If the same state is defined again as an accept state it will be modified only in case new token has higher priority.

Bool isAcceptState(int state):

It checks whether the given state is an accept state then returns true else return false.

```
void DFA::addAcceptState(int state,Token token) {
    if (accept_states.count(state)) {
        if (token.getPriority() < accept_states.at(state).getPriority()) {
            accept_states.at(state) = token;
        }
    } else {
        accept_states.insert( x pair<int, Token>(state, token));
    }
}

bool DFA::isAcceptState (int state) {
    if (accept_states.size() == 0) {
        return false;
    }
    map<int, Token>::iterator itr;
    for (itr =  accept_states.begin(); itr !=  accept_states.end(); ++itr) {
        if (itr->first == state) {
            return true;
        }
    }
    return false;
}
```

Bool areCompatibleStates(int state1, int state2, vector<int> states_partitions):

It checks if the 2 given states go to states of the same partition under each input then return true, else return false.

```
bool DFA::areCompatibleStates (int state1, int state2, vector<int> states_partitions) {
    for (int i = 0; i < number_of_inputs; i++) {
        if (table[state1][i] >= 0 && table[state2][i] >= 0) {
            if (states_partitions[table[state1][i]] != states_partitions[table[state2][i]]) {
                return false;
            }
        } else if ((table[state1][i]  < 0 && table[state2][i] >= 0) || ((table[state1][i] ) >= 0 && table[state2][i] < 0)){
            return false;
        }

    }
    return true;
}
```

DFA* minimize():

First it partitions all non accept states into one partition and every accept state into a partition.

Second it calculates the new state value for each state according to its partition.

Then we loop on the partitions and try to break them into smaller partitions if their composing states are not compatible, such that the final result is partitions of compatible states only.

This loop continues until the size of partitions can't be reduced anymore.

Finally create the new minimized DFA according to final partitions.

```
DFA* DFA::minimize() {
    vector< vector<int> > partitions;
    //partition states into one partition for all non-acceptance states
    // and one partition for each acceptance state.
    vector<int> nonAcceptStates;
    int counter = 0;
    vector <int> states_partitions( n: number_of_states,  value: -1);

    for (int i = 0; i < number_of_states; i++) {
        if (!isAcceptState(i)) {
            nonAcceptStates.push_back(i);
        } else {
            vector<int> v;
            v.push_back(i);
            partitions.push_back(v);
        }
    }
    partitions.insert(partitions.begin(),nonAcceptStates);

    for (int i = 0; i < partitions.size(); i++) {
        for (int j = 0; j < partitions[i].size(); j++) {
            states_partitions[partitions[i][j]] = i;
        }
    }
```

```
vector< vector<int> > old_partitions = partitions;
partitions.erase(partitions.begin(), partitions.end());
vector< vector<int> > new_partitions;
bool flag = true;
while (flag) {
    for (int i = 0; i < old_partitions.size(); i++) {
        //we are inside one partition here
        if (old_partitions[i].size() > 1) {
            //partition has more than one state
            // so it can be broken down into more states
            //put the 1st state of this partition into separate partition
            vector<int> v;
            v.push_back(old_partitions[i][0]);
            partitions.push_back(v);
            for (int j = 1; j < old_partitions[i].size(); j++) {
                bool isCompatiple = false;
                for (int k = 0; k < partitions.size(); k++) {
                    if (areCompatibleStates( state1: partitions[k][0], old_partitions[i][j], states_partitions)) {
                        partitions[k].push_back(old_partitions[i][j]);
                        isCompatiple = true;
                    }
                }
                if (!isCompatiple) {
                    vector<int> t;
                    t.push_back(old_partitions[i][j]);
                    partitions.push_back(t);
                }
            }
            //erase partitions vector to be filled by partitions of another broken one
            for (int k = 0; k < partitions.size(); k++) {
                new_partitions.push_back(partitions[k]);
            }
            partitions.erase(partitions.begin(), partitions.end());
        } else {
            //partition has one state so insert it in partitions
            new_partitions.push_back(old_partitions[i]);
        }
    }
    for (int i = 0; i < new_partitions.size(); i++) {
        for (int j = 0; j < new_partitions[i].size(); j++) {
            states_partitions[new_partitions[i][j]] = i;
        }
    }

    flag = (old_partitions.size() != new_partitions.size());
    old_partitions = new_partitions;
    new_partitions.erase(new_partitions.begin(), new_partitions.end());

}
```

```
if (number_of_states == old_partitions.size()) {
    return this;
}
DFA *dfa = new DFA(old_partitions.size(),  number_of_inputs: number_of_inputs);

//add transitions relative to new partitions
for (int i = 0; i < dfa->getNumberOfStates(); i++) {
    for (int j = 0; j < dfa->getNumberOfInputs(); j++) {
        if (table[ old_partitions[i][0] ] [j] != phai) {
            dfa->addTransition(i, j,  to: states_partitions[table[old_partitions[i][0]][j]]);
        }
    }
}

//add accept states relative to new partitions
map<int, Token>::iterator itr;
for (itr =  accept_states.begin(); itr != accept_states.end(); ++itr) {
    dfa->addAcceptState(states_partitions[itr->first], itr->second);
}

return dfa;
```

Void print_dfa():

It prints the minimized transition table into a text file “minimized transition table.txt” with a certain format.

```
void DFA::print_dfa() {
    if (number_of_states == 0)
        return;
    ofstream file;
    file.open( s: "minimized transition table.txt", std::ofstream::trunc);
    if (file.is_open()) {
        file << "state    ";
        cout << "state    ";
        for (int i = 0; i < number_of_inputs; i++) {
            char c = i + 32;
            file << " '" << c << "' ";
            cout << " '" << c << "' ";
        }
        file << endl;
        cout << endl;

        for (int i = 0; i < number_of_states; i++) {
            if (isAcceptState(i)) {
                if (i < 10) {
                    file << "   *" << i << "      ";
                    cout << "   *" << i << "      ";
                } else {
                    file << "  *" << i << "      ";
                    cout << "  *" << i << "      ";
                }
```

```
        } else {
            if (i < 10) {
                file << "    " << i << "     ";
                cout << "    " << i << "     ";
            } else {
                file << "   " << i << "     ";
                cout << "   " << i << "     ";
            }
        }
        for (int j = 0; j < number_of_inputs; j++) {
            if (table[i][j] < 0) {
                file << table[i][j] << "   ";
                cout << table[i][j] << "   ";
            } else {
                file << table[i][j] << "    ";
                cout << table[i][j] << "    ";
            }

        }
        file << endl;
        cout << endl;
    }
}
file.close();
```

NFA_constructor class

```
class NFA_constructor
{
    public:
        NFA_constructor();
        virtual ~NFA_constructor();
        NFA constructNFA(string expression);
        NFA kleene_closure(NFA original_nfa);
        NFA positive_closure(NFA original_nfa);
        NFA oring(NFA original1, NFA original2);
        NFA oringList(vector<NFA> list,bool combine);
        NFA concatinating(NFA original1, NFA original2);
        NFA signleCharNFA(vector<char> input);
        NFA termNFA(string term);
        void setRegular_definitions(vector<Regular_definition> regular_definitions);

    protected:

    private:
        string trim(string s);
        vector<Regular_definition> regular_definitions;
};

#endif // NFA_CONSTRUCTOR_H
```

Functions:

- **constructNFA**: A function that accepts a regular expression as a parameter and returns the NFA of this expression. The regular expression is parsed into subexpressions, then to its raw terms and characters to construct the final NFA based on Thompson's Construction Algorithm.
- **Kleene_closure**: A function that takes an NFA and returns a new NFA that is the kleene closure NFA of it.
- **positive_closure**: A function that takes an NFA and returns a new NFA that is the positive closure NFA of it.
- **oring**: A function that takes two NFAs and returns a new NFA that is the result of ORing these two NFAs.
- **oringList**: A function that takes a list of NFAs and performs OR on them. The function is used in two scenarios, the first is when we want to simply OR the NFAs with new start and accept states. The second is when we want to combine NFAs with different accept states and different tokens. This is determined by the combine boolean.
- **concatenating**: A function that takes two NFAs, concatenates them together and returns the new concatenated NFA.
- **singleCharNFA**: A function that takes a vector of inputs and builds a simple two state NFA for it using this input.
- **termNFA**: A function that takes a string term and returns the constructed NFA for this term.

Lexical_Analyzer Class

```
class Lexical_analyzer {
    public:
        Lexical_analyzer();
        virtual ~Lexical_analyzer();
        void execute(string file_name);
        void read_input(string file_name);
        void setDFA(DFA *dfa);
        void analyze (vector<char> input_code);
        void set_input_code (vector<char> input_code);

    private:
        vector<char> input_code;
        vector<Token> tokens;
        vector<string> symbol_table;
        DFA *dfa;

};
```

Void read_input(string file_name):

It reads the input file and converts it into a vector of characters.

Void analyze(vector<char> input_code):

It takes the vector of characters.

initially starting from state 0, it moves according to the sequence of characters in the vector.

If an accept state is reached it saves the accepted token, the current state and the index of the current character in the vector.

If phai state or the last character or a space is reached it checks if there was an accepted state, if so it prints the accepted output.

If it was an id it saves it in a symbol table.

Else If a phai state is reached and the current character is not a space it prints that a lexical error has occurred.

Then return back to state 0 again to start in analyzing a new token.

Void execute(string file_name):

It calls on read_input() then analyze() functions respectively.

```
void Lexical_analyzer::execute(string file_name) {
    read_input(file_name);
    analyze( input_code: input_code);
}

void Lexical_analyzer::read_input(string file_name) {
    vector<char> input_code;
    ifstream file;
    file.open(file_name, ios::in);
    if (file.is_open()) {
        for (char c; file.get( &: c);) {
            if (c != '\n') {
                input_code.push_back(c);
            }
        }
    }
    set_input_code(input_code);
}
```

```
void Lexical_analyzer::analyze(vector<char> input_code) {
    int current_state = 0;
    int input;
    //info about the total analyzer state when last match occured
    // they have to be reset when state is accepted by resetting last_accepted_output to empty character
    string last_accepted_output = "";
    int last_accepted_state = 0;
    int last_accepted_character_index;
    string id = "";
    int id_counter = 0;
    char c;
    int i = 0;
    while (i < input_code.size()) {
        c = input_code[i];
        id += c;
        input = c - 32;
        current_state = dfa->getTable()[current_state][input];
        if (dfa->isAcceptState(current_state)) {
            last_accepted_state = current_state;
            last_accepted_character_index = i;
            if (dfa->getAcceptStates()[current_state].getToken_class() == "keyword"
            || dfa->getAcceptStates()[current_state].getToken_class() == "punctuation") {
                last_accepted_output = dfa->getAcceptStates()[current_state].getValue();
            } else {
                last_accepted_output = dfa->getAcceptStates()[current_state].getToken_class();
            }
            // add the recognized token to the vector of tokens
            tokens.push_back(dfa->getAcceptStates()[current_state]);
        }
```

```
if (current_state == phai || c == 32 || i == input_code.size()-1) {
    if (last_accepted_output != "") {
        id = id.substr( pos: 0,  n: id.size()-(i-last_accepted_character_index));
        cout << id << " --> " << last_accepted_output << endl;
        //add the matched ids to a symbol table
        if (dfa->getAcceptStates()[current_state].getToken_class() == "id") {
            Token *t = new Token();
            t->setToken_class( token_class: "id");
            //remove last (i - last_accepted_character_index) characters which are extra than the last match
            t->setValue(id);
            symbol_table.push_back(dfa->getAcceptStates()[current_state].getValue());
        }
        last_accepted_output = "";
        //backtrack to last matched state and character
        i = last_accepted_character_index;
        if (c == 32) {
            i++;
        }

    } else {
        //If no matches happened and phai state reached and current character is not a space so error occured
        if (c != 32) {
            cout << c << " --> " << "Lexical error" << endl;
        }
    }
    //reset state again to start searching for tokens
    id = "";
    current_state = 0;
}
i++;
```

Lexical_Analyzer_Generator Class

Lexical_Analyzer_Generator class is responsible for controlling the flow of generating the lexical analyzer from the set of the given rules. It mainly has 4 vector attributes representing the list of keywords, punctuations, regular definitions and regular expressions extracted from the input rules. Also, it has a pointer to the generated minimal DFA.

```
#ifndef LEXICAL_ANALYZER_GENERATOR_H
#define LEXICAL_ANALYZER_GENERATOR_H
#include "Token.h"
#include "NFA.h"
#include "DFA.h"
#include"Regular_definition.h"
#include"Regular_expression.h"
#include<string>
#include <vector>

using namespace std;

class Lexical_analyzer_generator
{
    public:
        Lexical_analyzer_generator();
        virtual ~Lexical_analyzer_generator();
        void read_lexical_rules(string file_name);
        void generate_lexical_analyzer();
        DFA *get_minimal_dfa();

    protected:

    private:
        void classify_line(string line, int priority);
        vector<Regular_expression> regular_expressions;
        vector<Regular_definition> regular_definitions;
        vector<pair<string, int>> keywords;
        vector<pair<string, int>> punctuations;
        string trim(string s);
        DFA *minimal_dfa;
};
```

read_lexical_rules()

The function reads lexical rules line by line from the specified input file, for each line the helping function “classify_line” is called passing the current priority.

```
void Lexical_analyzer_generator::read_lexical_rules(string file_name)
{
    ///read rules line by line from lexical rules file, specify line class and set priority
    fstream file;
    file.open(file_name, ios::in);
    if (file.is_open())
    {
        int priority = 1;
        string line;
        while (getline(file, line))
        {
            classify_line(line, priority);
            priority++;
        }
    }
}
```

classify_line()

The function accepts a string line and int priority parameters then adds the given line to one of the four vectors according to its class: keyword, punctuation, regular definition and regular expression. Classification is based on the defined rules format.

```
void Lexical_analyzer_generator::classify_line(string line, int priority)
{
    line = trim(line);
    ///keywords
    if (line.at(0) == '{')
    {
        line = line.substr(1, line.length()-2);
        line = trim(line);
        istringstream ss(line);
        while (ss)
        {
            string keyword;
            ss >> keyword;
            if (keyword != "")
            {
                pair<string,int> p(keyword, priority);
                keywords.push_back(p);
            }
        }
        return;
    }
    ///Punctuations
    if (line.at(0) == '[')
    {
        line = line.substr(1, line.length()-2);
        line = trim(line);
        istringstream ss(line);
        while (ss)
        {
            string punc;
            ss >> punc;
            if (punc != "")
            {
                if (punc.length() > 1 && punc.at(0)=='\\')
                {
```

```
                punc = punc.substr(1);
            }
            pair<string,int> p(punc, priority);
            punctuations.push_back(p);
        }
    }
    return;
}
///Regular definitions
size_t f = line.find("=");
if (f != std::string::npos)
{
    if (line.at(f-1) != '\\')
    {
        Regular_definition *rd = new Regular_definition();
        string name = trim(line.substr(0,f));
        string value = trim(line.substr(f+1));
        rd->setName(name);
        rd->setValue(value);
        regular_definitions.push_back(*rd);
        return;
    }
}
///Regular expressions
f = line.find(":");
if (f != std::string::npos)
{
    if (line.at(f-1) != '\\')
    {
        Regular_expression *re = new Regular_expression();
        string name = trim(line.substr(0,f));
        string value = trim(line.substr(f+1));
        re->setName(name);
        re->setValue(value);
```

generate_lexical_analyzer()

Firstly, the function constructs an NFA to each keyword, punctuation symbol and regular expression. Each constructed NFA is added to the NFA list, then all NFAs in the list are combined together. The combined NFA is converted to DFA which is then minimized. The minimal_dfa pointer is set to point to this final DFA.

```
void Lexical_analyzer_generator::generate_lexical_analyzer()
{
    vector<NFA> NFAlist;
    NFA_constructor *constructor = new NFA_constructor();
    constructor->setRegular_definitions(regular_definitions);
    ///construct NFA for each keyword
    for (unsigned int i = 0; i < keywords.size(); i++)
    {
        NFA nfa = constructor->termNFA(keywords[i].first);
        Token *token = new Token();
        token->setToken_class("keyword");
        token->setValue(keywords[i].first);
        token->setPriority(keywords[i].second);
        int accept = nfa.getAcceptState();
        nfa.addAcceptStateToList(accept, *token);
        NFAlist.push_back(nfa);
    }
    ///construct NFA for each punctuation character
    for (unsigned int i = 0; i < punctuations.size(); i++)
    {
        string punc = punctuations[i].first;
        vector<char> vec;
        vec.push_back(punc.at(0));
        NFA nfa = constructor->signleCharNFA(vec);
        Token *token = new Token();
        token->setToken_class("punctuation");
        token->setValue(punc);
        token->setPriority(punctuations[i].second);
        int accept = nfa.getAcceptState();
        nfa.addAcceptStateToList(accept, *token);
        NFAlist.push_back(nfa);
    }
    ///get constructed NFA of each regular expression
    for (unsigned int i = 0; i < regular_expressions.size(); i++)
    {
        NFA nfa = regular_expressions[i].getNFA(regular_definitions);
        NFAlist.push_back(nfa);
    }
    ///combine all NFA
    NFA combined = constructor->oringList(NFAlist, true);
    ///convert NFA to DFA
    DFA* dfa = combined.convertToDFA();
    ///minimize DFA
    minimal_dfa = dfa->minimize();
    minimal_dfa->print_dfa();
}
```

get_minimal_dfa()

The function returns a pointer to the minimal constructed DFA.

Main

```
int main()
{
    Lexical_analyzer_generator *generator = new Lexical_analyzer_generator();
    generator->read_lexical_rules( file_name: "rules.txt");
    generator->generate_lexical_analyzer();
    Lexical_analyzer *lexical = new Lexical_analyzer();
    lexical->setDFA(generator->get_minimal_dfa());
    lexical->execute( file_name: "input.txt");

    return 0;
}
```

Create a new lexical analyzer generator.
Make it read the rules in the “rules.txt” file to create the transition table of these rules.
Create a new lexical analyzer passing to it the minimized dfa.
Call the execute function on the input code in the “input.txt” file.

Parser Generator Design

Symbol Class:

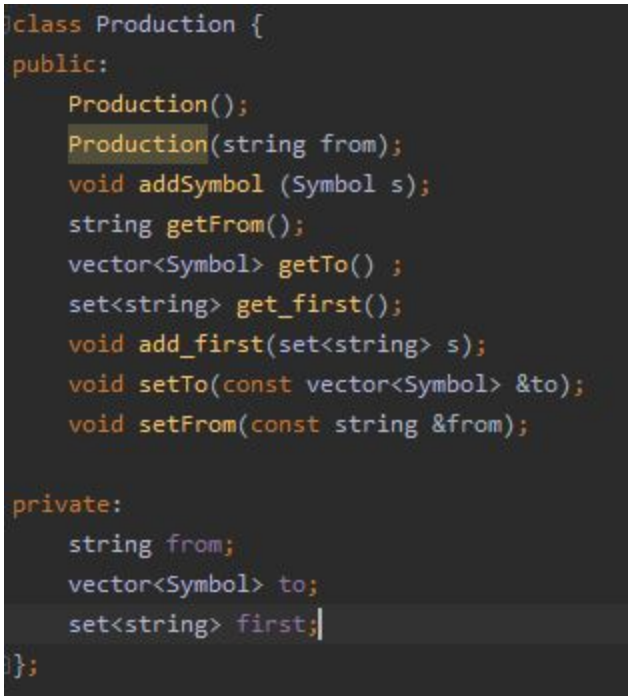
```
class Symbol {
public:
    Symbol();
    Symbol(string symbol, bool isTerminal);
    string getSymbol();
    bool isTerminal();
    void setSymbol(const string &symbol);
    void setIsTerminal(bool isTerminal);

private:
    string symbol;
    bool is_terminal;
};
```

A class that creates a symbol for either a terminal or a non-terminal that contains:

String symbol: contains the string value of the symbol.

Bool is_terminal: is true if terminal else it is false.

Production Class:



```
class Production {
public:
    Production();
    Production(string from);
    void addSymbol (Symbol s);
    string getFrom();
    vector<Symbol> getTo() ;
    set<string> get_first();
    void add_first(set<string> s);
    void setTo(const vector<Symbol> &to);
    void setFrom(const string &from);

private:
    string from;
    vector<Symbol> to;
    set<string> first;
};
```

A class that creates a production that contains:

string from: which carries the non-terminal symbol on the left hand side of the production.

Vector <Symbol> to: which carries the symbols that this production goes to, which are on the right hand side of it.

Set <string> first: set of strings representing the set of first terminals of the production.

addSymbol(Symbol s): adds a symbol to the “to” vector.

addFirst(set <string> s): adds set of terminals to the first set of the production.

Parser Table Class:

```
class Parser_table {
public:
    Parser_table();
    Parser_table(map <string, int> non_terminals, map <string, int> terminals, map<int, vector<Production>> productions,
           map<int, set<string>> first_sets, map<int, set<string>> follow_sets);
    void addTerminal (Production p);
    void addNonTerminal (Production p);
    Production **getTable();
    map<string, int> getTerminals();
    map<string, int> getNonTerminals();
    void setTerminals(map<string, int> terminals);
    void setNonTerminals(map<string, int> nonTerminals);
    void printTable();
    string printHelper(const string x, const int width);
    Parser_table build_table();

private:
    Production **table;
    map <string, int> terminals;
    map <string, int> non_terminals;
    map<int, set<string>> first_sets;
    map<int, set<string>> follow_sets;
    map<int, vector<Production>> productions;
};
```

A class that creates a parser table that the parser will be using in parsing tokens later that contains:

Production **table: which is a 2-D array of productions that helps the parser to know which production does it go to from a non-terminal under each terminal.

Map <string, int> terminals: a map that carries the index of each terminal in the parser table.

Map <string, int> non-terminals: a map that carries the index of each non-terminal in the parser table.

Map<int, set<string>> first_sets: contains the first set of each non terminal.

Map<int, set<string>> follow_sets: contains the follow set of each non terminal.

map<int, vector<Production>> productions: contains the productions of each non terminal.

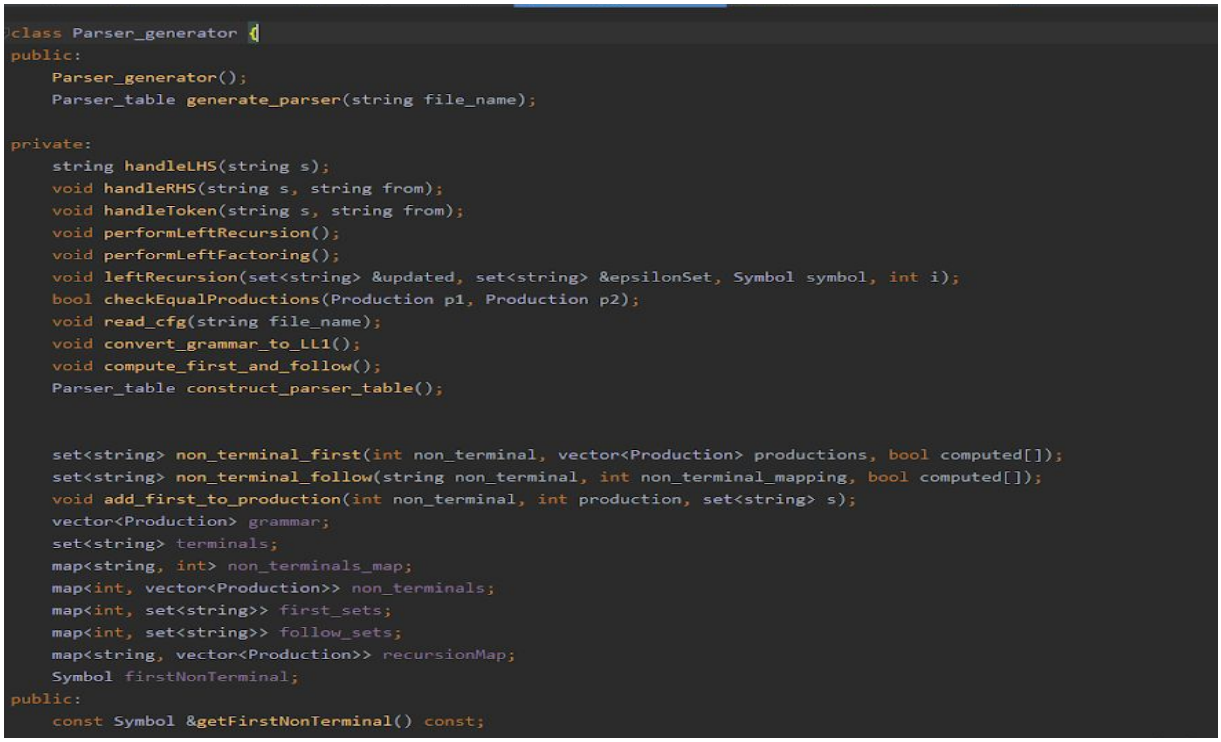
printTable(): a method that helps print the parser table in a readable format in the file called “parser_table.txt”.

String printHelper(string x, int width): a print helper that helps the printTable method in printing the table in a readable format by giving each print “x” a specific width “width”.

Parser_table build_table(): builds the table with productions for each non terminal under each terminal using the first and follow sets.

```
Parser_table Parser_table::build_table(){
    map<string,int>::iterator it = non_terminals.begin();
    while(it!=non_terminals.end()){
        int i = it->second;
        string s = it->first;
        set<string> follow = follow_sets[i];
        set<string> first = first_sets[i];
        vector<Production> pro = productions[i];
        set<string>::iterator follow_it = follow.begin();
        Symbol *synch = new Symbol("synch", true);
        Symbol *eps = new Symbol("epsilon", true);
        if(first.count("epsilon")){
            Production *p = new Production(s);
            p->addSymbol(*eps);
            while(follow_it!=follow.end()){
                table[i][terminals[*follow_it]] = *p;
                follow_it++;
            }
        }else{
            Production *p = new Production(s);
            p->addSymbol(*synch);
            while(follow_it!=follow.end()){
                table[i][terminals[*follow_it]] = *p;
                follow_it++;
            }
        }
        for(int j=0;j<pro.size();j++){
            Production p = pro[j];
            set<string> f = p.get_first();
            set<string>::iterator first_it = f.begin();
            while(first_it!=f.end()){
                table[i][terminals[*first_it]] = p;
                first_it++;
            }
        }
        it++;
    }
    return *this;
}
```

Parser Generator Class:



```
class Parser_generator {
public:
    Parser_generator();
    Parser_table generate_parser(string file_name);

private:
    string handleLHS(string s);
    void handleRHS(string s, string from);
    void handleToken(string s, string from);
    void performLeftRecursion();
    void performLeftFactoring();
    void leftRecursion(set<string> &updated, set<string> &epsilonSet, Symbol symbol, int i);
    bool checkEqualProductions(Production p1, Production p2);
    void read_cfg(string file_name);
    void convert_grammar_to_LL1();
    void compute_first_and_follow();
    Parser_table construct_parser_table();


    set<string> non_terminal_first(int non_terminal, vector<Production> productions, bool computed[]);
    set<string> non_terminal_follow(string non_terminal, int non_terminal_mapping, bool computed[]);
    void add_first_to_production(int non_terminal, int production, set<string> s);
    vector<Production> grammar;
    set<string> terminals;
    map<string, int> non_terminals_map;
    map<int, vector<Production>> non_terminals;
    map<int, set<string>> first_sets;
    map<int, set<string>> follow_sets;
    map<string, vector<Production>> recursionMap;
    Symbol firstNonTerminal;
public:
    const Symbol &getFirstNonTerminal() const;
```

generate_parser(string file_name): A function that takes the CFG file name and calls other functions to read the grammar, transform it into LL1, compute the first and the follow then generate the parser table.

read_cfg(string file_name): A function that takes the CFG file name as input, reads the file and parses it. The function generates the initial grammar without eliminating the left recursion or left factoring.
The function uses the following helper functions in the process of prasing..

1) handleLHS(string s): the function handles parsing the left hand side of the production rule to extract the non terminal.
2) handleRHS(string s, string from): the function handles parsing the right hand side of the production rule to extract the terminals and the non terminals. It is also responsible for creating the production rule and putting it in the grammar vector.

covert_grammar_to_LL1( ): the function is responsible for converting the current grammar vector of productions into a LL1 productions which is done in two steps...

1) performLeftRecursion( ): the function removes left recursions from the current set of productions. It generates a new grammar vector with productions that are left recursion free.

2) performLeftFactoring( ): the function loops through the current set of productions and removes the left factoring if found. The function puts the new productions into the following two maps

    a) Non_terminals_map: the key of the map is the value of the non terminal. The key is the index of this non terminal in the parser table.

    b) Non_terminals: the key of the map is the index of the non terminal in the parser table. The value is a vector of production with this non terminal as the LHS.

compute_first_and_follow(): the function first loops on all productions to compute the first set of each non-terminal. Then it loops on all non-terminals to compute their follow sets. First and follow sets are saved in two maps:

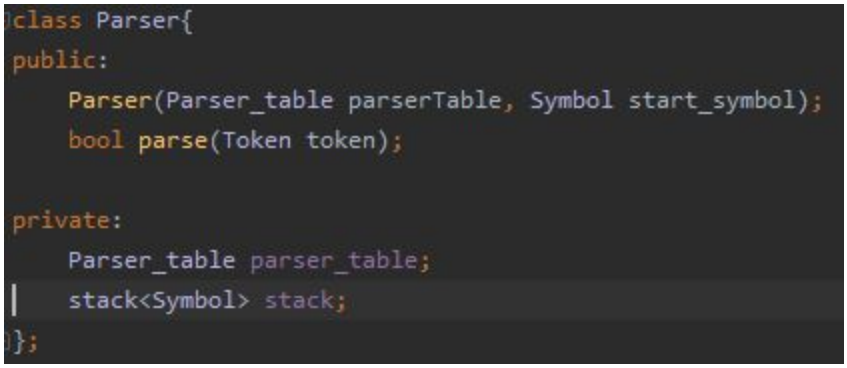
- first_sets: map of integer key and set<string> value, where the integer key represents the mapping of each non-terminal and the set of strings represents the set of its first set of terminals.
- follow_sets: similar to first_sets map, but the set of strings represents the follow set of non-terminal.

non_terminal_first(): the function accepts three parameters: integer non terminal, which represents the mapping of the non-terminal to compute the first set for, vector<Production>, that carries the productions having this non-terminal in their LHSs, and a bool array computed, to indicate which of the non terminals has computed first set. The function loops on the vector of productions, for each production if it starts with a terminal then it is added to the first set, otherwise a loop is occured on the symbols of the production as long as the symbol is a non-terminal and has an epsilon in its first set. For each non-terminal symbol, it is checked whether its first set has been computed, if so then its elements are added to the current first set, otherwise the function is called recursively to compute the first set of the current non-terminal and add its elements. The function also updates the first set of each production during computation. Finally it updates the computed array and the first_sets map and returns a set of strings representing the first set.

non_terminal_follow(): the function accepts three parameters, a string representing the non-terminal to compute the follow set for, an integer representing its mapping and a boolean array computed to indicate which non-terminals have computed follow sets. The function loop on all non-terminals and their productions, for each production, if it has the non-terminal in its RHS followed by a terminal then this terminal is added to its follow set. And if it is followed by a non-terminal then the first set elements of each following non-terminal are added to the follow set, except for the empty string, as long as these non terminals have empty string in their first sets. Also for each production with LHS other than the current non-terminal, if the current non-terminal is the last symbol or it is followed by non-terminals that have empty strings in their first sets, it is checked if the follow set of the LHS terminal has been computed, if so then in its elements are added to the current follow set, otherwise the function is called recursively. Finally the computed array and follow_sets map are updated and a set of strings representing the follow set is returned.

add_first_to_production(): the function takes three parameters, the integer mapping of a non-terminal, an integer index of the production and a set of strings. The set of strings are added to the first set of the specified production.

Parser Class:



```
class Parser{
public:
    Parser(Parser_table parserTable, Symbol start_symbol);
    bool parse(Token token);

private:
    Parser_table parser_table;
    stack<Symbol> stack;
};
```

A class that does the parsing through having the starting non-terminal symbol, the parser table and a stack.
Parser(Parser_table parser_table, Symbol start_symbol): creates a parser by setting its parser table and pushing the '$' in the stack then the starting symbol.

parse(Token token): given a token by calling the getNextToken() through the lexical analyzer it parses it such that while the top symbol of the stack is a non-terminal the following steps are done:

- Pop the top non-terminal from stack.
- Get the production that it goes to under the current token.
- If it goes to error, return the symbol to stack again and continue.
- If it goes to synch, continue ignoring the popped symbol.
- If it goes to epsilon, continue ignoring the popped symbol.
- Else push all the symbols in the RHS of the production from right to left.

When the loop ends it starts matching the terminal found as follows :

- Pop the current terminal from the stack.
- If matching with current token failed:
    - If the stack is empty then parsing fails.
    - Else we just ignore the current non-matched non-terminal
- Else when matching succeeds if it's a '$' symbol then parsing is done, else it goes for the next token if found.

Lexical Analyzer Class:

```
class Lexical_analyzer {
    public:
        Lexical_analyzer();
        virtual ~Lexical_analyzer();
        void execute(string file_name);
        void read_input(string file_name);
        void setDFA(DFA *dfa);
        void analyze (vector<char> input_code);
        void set_input_code (vector<char> input_code);
        Token getNextToken ();

    private:
        vector<char> input_code;
        vector<Token> tokens;
        vector<string> symbol_table;
        int current_token_index;
        DFA *dfa;
};
```

```
Token Lexical_analyzer::getNextToken () {
    if (current_token_index != tokens.size()) {
        Token token = tokens.at( n: current_token_index);
        current_token_index++;
        return token;
    } else {
        Token *token = new Token();
        token->setValue( value: "$");
        return *token;
    }
}
```

Function Token getNextToken() added to the lexical analyzer class such that:

- It helps in integrating the lexical analyzer phase with the parser phase.
- It returns the next token from the vector of tokens each time it's called.
- After the whole vector is returned it returns a token of the '$' symbol such that the parser can detect that tokens are finished.

Main:

```
int main() {
    Lexical_analyzer_generator *generator = new Lexical_analyzer_generator();
    generator->read_lexical_rules( file_name: "rules.txt");
    generator->generate_lexical_analyzer();
    Lexical_analyzer *lexical = new Lexical_analyzer();
    lexical->setDFA(generator->get_minimal_dfa());
    lexical->execute( file_name: "input.txt");

    Parser_generator *parserGenerator = new Parser_generator();
    Parser_table parserTable = parserGenerator->generate_parser( file_name: "cfg.txt");
    parserTable.build_table().printTable();

    Parser *myParser = new Parser(parserTable.build_table(), parserGenerator->getFirstNonTerminal());
    Token t = lexical->getNextToken();
    bool error_flag = false;
    while (t.getValue() != "$") {
        if (myParser->parse(t)) {
            t = lexical->getNextToken();
        } else {
            error_flag = true;
            break;
        }
    }
    while (!error_flag) {
        error_flag = !myParser->parse(t);
    }
    return 0;
```

- After running the lexical analyzer on the given input code file and building the list of tokens.
- Parser generator is run on the given grammar in the “cfg.txt” file.
- After that the build_table() function is called.
- Then create a new parser.
- Try to get the next token from the lexical analyzer if found the parser should parse it.
- Repeat this loop till it passes through all the tokens or till parsing fails.

Java Byte Code Design

Semantic rules (bison.y) file

```
%{
#include <unistd.h>
#include "stdc++.h"

using namespace std;

extern int yylex();
extern FILE *yyin;
void yyerror(const char * s);

int id_counter = 1;
typedef enum {INT_TYPE, FLOAT_TYPE} type;
map<string, pair<int,type>> symbol_table;
ofstream outFile("output.txt");

vector<string> javaByteCode;
vector<int> *make_list(int index);
vector<int> *merge(vector<int> *p1, vector<int> *p2);
void back_patch(vector<int> *p, int index);
void declare_variable (string id_str, int id_type);
void addLine(string s);
void print_output();
bool is_valid_id(string id);
%}
```

Int Id_counter: represents the number of identified ids

Enum Type: enum for determining the type of id.

map<string, <int, type>> Symbol_table: maps the id string with a pair, the first is its index and the second is its enum type (INT_TYPE, FLOAT_TYPE).

Vector<string> javaByteCode: a vector that holds the generated java byte code.

```
//makelist --> creates and returns a new list that only contains an index to an instruction
vector<int> *make_list(int index)
{
  vector<int> *vec = new vector<int>();
  vec->push_back(index);
  return vec;
}
```

vector<int> *make_list(index): creates a new list containing only index, an index into the array of instructions; makelist returns a pointer to the newly created list.

```
//merge --> concatenates p1 and p2 and returns the concatenated list
vector<int> *merge(vector<int> *p1, vector<int> *p2)
{
  if(p1 == NULL && p2 == NULL){
    vector<int> *vec = new vector<int>();
    count<<"both empty"<<endl;
    return vec;
  }
  else if(p1 == NULL){
  count<<"1 is empty"<<endl;
    return p2;
  }
  else if(p2 == NULL){
  count<<"2 is empty"<<endl;
    return p1;
  }
  else{
    count<<"before vec"<<endl;
    vector<int> *vec = new vector<int>(*p1);
    count<<"after vec"<<endl;
    for(int i=0; i<p2->size();i++){
        vec->push_back((*p2)[i]);
    }
    count<<"after merge"<<endl;
    return vec;
  }
}
```

vector<int> *merge(vector<int> *p1, vector<int> *p2): concatenates the lists pointed to by pl and p2 , and returns a pointer to the concatenated list.

```
//backpatch --> inserts index as target label for each instruction in p
void back_patch(vector<int> *p, int index)
{
  if(p == NULL){
    return;
  }
  for(int i=0; i<p->size(); i++){
    int int_val = (*p)[i];
    javaByteCode[int_val] = javaByteCode[int_val] + to_string(index);
  }
}
```

void back_patch(vector<int> *p, int index): inserts index as the target label for each of the instructions on the list pointed to by p.

```
void declare_variable (string id_str, int id_type)
{
        if (is_valid_id(id_str)) {
                string error = id_str + " was declared before";
                yyerror(error.c_str());
        } else {
                if (id_type == INT_TYPE) {
                        addLine("iconst_0\nistore " + to_string(id_counter));
                } else if (id_type == FLOAT_TYPE) {
                        addLine("iconst_0\nistore " + to_string(id_counter));
                }
                symbol_table[id_str] = make_pair (id_counter++, (type)id_type);
        }
}
```

void declare_variable (string id_str, int id_type): adds a new id to the symbol table if not identified before.

```
//adds a new line in the javaByteCode list
void addLine(string s)
{
  javaByteCode.push_back(s);
}

//checks if the id is already identified or not
bool is_valid_id(string id)
{
  return (symbol_table.find(id) != symbol_table.end());
}
```

void addLine(string s): adds a java bytecode line to the **javaByteCode** vector.

bool is_valid_id(string id): checks if the id is already identified or not.

```
//print java byte code
void print_output()
{
  for ( int i = 0 ; i < javaByteCode.size() ; i++)
  {
    outFile<<javaByteCode[i]<<endl;
  }
}
```

void print_output(): prints all the generated JBC in the output.txt file.

```
void yyerror(const char * s)
{
 fprintf (stderr, "%s\n", s);
}
```

void yyerror(const char * s): prints out the string error s.

```
%code requires {
        #include <vector>
        #include <map>
        using namespace std;
}

%start method_body
```

```
%union{
        int int_val;
        float float_val;
        char * id_val;
        struct {
                vector<int> *true_list;
                vector<int> *false_list;
        } bool_expression;
        struct {
                int type;
        } expression_type;
        int id_type;
        char* operation;
        bool boolean_val;
        struct {
        vector<int> *next_list;
        } statement_type;
}
```

```
%token <id_val> identifier
%token <int_val> int_value
%token <float_val> float_value
%token <boolean_val> boolean
%token <operation> relop
%token <operation> boolean_op //and, or , not
%token semi_colon
%token equals
%token int_word
%token float_word
%token boolean_word
%token <operation> mulop
%token <operation> addop

%type <id_type> primitive_type
%type <bool_expression> boolean_expression
%type <expression_type> expression
%type <expression_type> simple_expression
%type <expression_type> term
%type <expression_type> factor
%type <statement_type> statement_list
%type <statement_type> statement
%type <statement_type> while
%type <statement_type> if
%type <statement_type> N
%type <int_val> M
%type <operation> sign
```

```
method_body: statement_list;

statement_list: statement_list statement | statement;

statement: declaration
| if {$$.next_list = $1.next_list;}
| while {$$.next_list = $1.next_list;}
| assignment;
```

 # METHOD_BODY = STATEMENT_LIST

 # STATEMENT_LIST = STATEMENT | STATEMENT_LIST STATEMENT

 # STATEMENT = DECLARATION | IF | WHILE | ASSIGNMENT

The action sets the next_list of statement to either a new empty vector in case of DECLARATION and ASSIGNMENT or sets it to the next_list of IF and WHILE**.**

```
assignment:
        identifier equals expression semi_colon {
                string id_str($1);
                if(is_valid_id(id_str)){
                        if($3.type == symbol_table[id_str].second) {
                                if($3.type == INT_TYPE) {
                                        addLine("istore " + to_string(symbol_table[id_str].first));
                                }
                                else if ($3.type == FLOAT_TYPE) {
                                        addLine("fstore " + to_string(symbol_table[id_str].first));
                                }
                        } else {
                                string error = "identifier and expression are not of the same type";
                                yyerror(error.c_str());
                        }
                } else {
                        string error = id_str + " isn't declared in this scope";
                        yyerror(error.c_str());
                }
        };

expression: simple_expression
{$$.type = $1.type;};
```

 # ASSIGNMENT = ‘id’ ‘=’ EXPRESSION ‘;’

If the id exists already then throw an error, else check if the type of the id and the expression if they are of different type then there is an error, else add a line to the java byte code list with istore/fstore and the index of this id in the symbol table.

 # EXPRESSION = SIMPLE_EXPRESSION

Sets the type of the RHS to the type of the simple expression

```
primitive_type:
        int_word {$$ = INT_TYPE;}
        | float_word {$$ = FLOAT_TYPE;};

declaration:
        primitive_type identifier semi_colon {
                string id_str($2);
                if($1 == INT_TYPE){
                        declare_variable (id_str, INT_TYPE);
                } else if ($1 == FLOAT_TYPE){
                        declare_variable (id_str, FLOAT_TYPE);
                }
        };
```

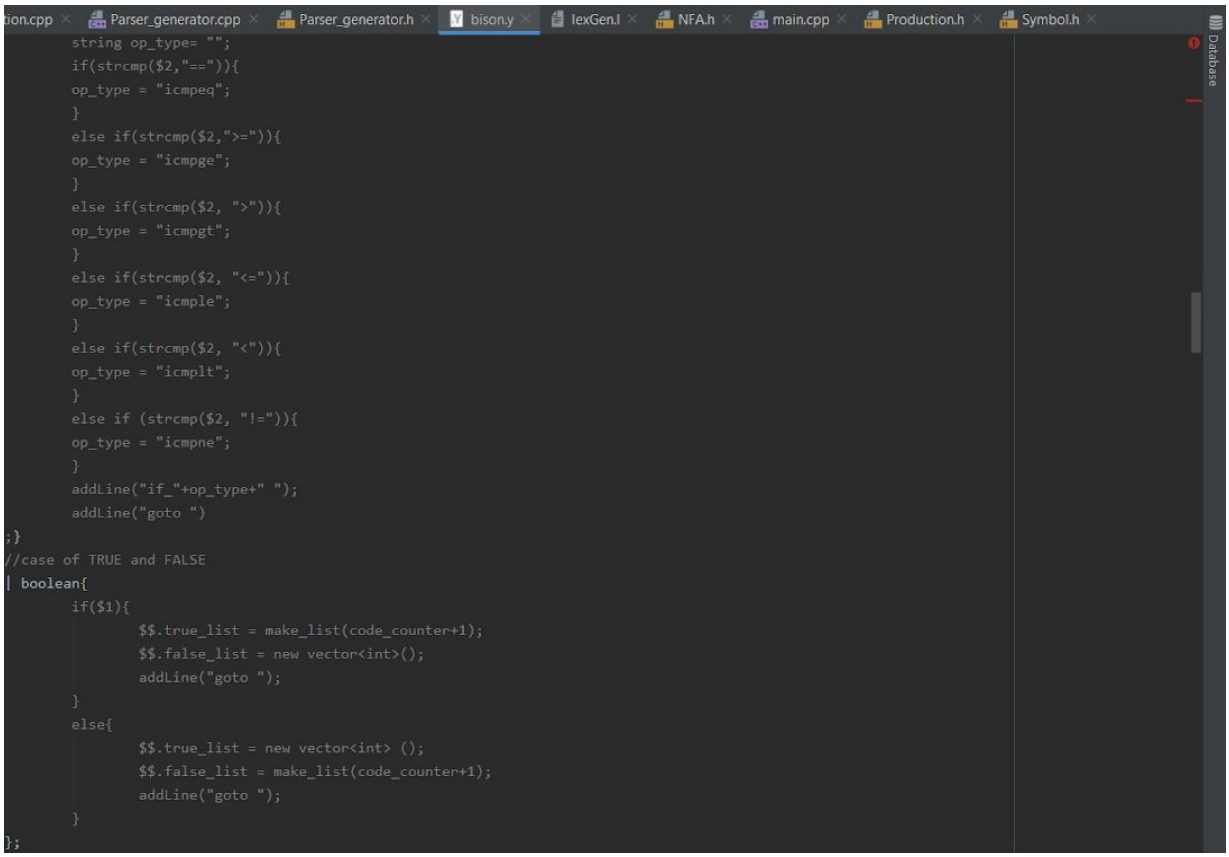
 # PRIMITIVE_TYPE = ‘int’ | ‘float’

Sets the type of the word with its enum type (INT_TYPE, FLOAT_TYPE)

 # DECLARATION = PRIMITIVE_TYPE ‘id’ ‘;’

Declare a new id in the symbol table with its type if not identified before.

```
boolean_expression :
//case of AND, OR
 boolean_expression boolean_op M boolean_expression {  //create label is still not ready
        if(strcmp($2,"||") == 0){
                back_patch($1.false_list, $3);
                $$.true_list = merge($1.true_list, $4.true_list);
                $$.false_list = $4.false_list;
        }
        else if(strcmp($2,"&&") == 0){
                back_patch($1.true_list,$3);
                $$.true_list = $4.true_list;
                $$.false_list = merge($1.false_list, $4.false_list);
        }
}
//case of RELOP
| expression relop expression {
        $$.true_list = make_list(code_counter);
        $$.false_list = make_list(code_counter+1);
        string op_type= "";
        if(strcmp($2,"==")){
        op_type = "icmpeq";
        }
        else if(strcmp($2,">=")){
        op_type = "icmpge";
        }
        else if(strcmp($2, ">")){
        op_type = "icmpgt";
        }
        else if(strcmp($2, "<=")){
        op_type = "icmple";
        }
        else if(strcmp($2, "<")){
        op_type = "icmplt";
        }
        else if (strcmp($2, "!=")){
```



 # boolean_expression = boolean_expression boolean_op boolean_expression

Describes the case of OR/AND where in the case of OR, we backpatch the first boolean expression false list with the next instruction which executes the second boolean expression then we set the RHS next list with the output of merging the true lists of both boolean expressions. Then we set the false list of RHS to the false list of the second boolean expression. In the case of AND we backpatch the first boolean expression true list with the next instruction. We set the true list of RHS to the next list of the second boolean expression and we set the false list of RHS to the merged list containing the false lists of both boolean expressions.

 # boolean_expression = expression relop expression

We set the true list of RHS to a newly created list that contains the next instruction. We also set the false list of RHS to another new list containing the following instruction. We identify the relative operator and we add a new line with it in the java byte code and add a “goto” to the java byte code.

 # boolean_expression = boolean

In case of TRUE, we set the true list of RHS to a newly created list with the next instruction and add a “goto” line to the java byte code. Similarly, in case of FALSE, we set the false list of

RHS to a newly created list with the next instruction and add a “goto” line to the java byte code.

```
while: while_token round_open M boolean_expression round_close curly_open M statement curly_close
{
        back_patch($4.true_list, $7);
        //back_patch($4.false_list, code_counter+2);
        back_patch($8.next_list, $3);
        $$.next_list = $4.false_list;
        addLine("goto " + to_string($3));
};

if: if_term round_open boolean_expression round_close curly_open M statement curly_close{
cout<<1<<endl;
back_patch($3.true_list, $6);
cout<<2<<endl;
$$.next_list = merge($3.false_list,$7.next_list);
cout<<3<<endl;
}
| if_term round_open boolean_expression round_close curly_open M statement curly_close N else_term curly_open M statement curly_close
{
back_patch($3.true_list,$6);
back_patch($3.false_list,$12);
vector<int> *temp = new vector<int>();
temp = merge($7.next_list,$9.next_list);
$$.next_list = merge(temp,$13.next_list);
};
```

 # WHILE = ‘while’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATEMENT ‘}’

Back patches the boolean expression true list with the next instruction which makes it executes the statement. It also back patches the statement next_list with the first M instruction, and adds a “goto” line with the same instruction which makes it go back to the boolean expression. The next list of RHS is set to the false list of the boolean expression.

 # IF = ‘if’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATEMENT ‘}’ ‘else’ ‘{‘ STATEMENT ‘}’

Back patches the true_list of the boolean expression with M (the next instruction in case of True) so it starts executing the first statement. It does the same with the false_list of the boolean expression and the second M so that it executes the second statement. It then merges the next lists of both statements and N, a production that goes to Epsilon, and adds a goto line to the java byte code and sets the next list of the RHS to the merged list.

```
N: {
$$.next_list = make_list(javaByteCode.size());
addLine("goto ");
};
```

```
simple_expression:
term {$$.type = $1.type;}
| sign term {
        $$.type = $2.type;
        if (strcmp($1,"-")==0) {
                if ($2.type == INT_TYPE) {
                        addLine("ineg");
                } else {
                        addLine("fneg");
                }
        }
}
| simple_expression addop term {
        if ($1.type == FLOAT_TYPE || $3.type == FLOAT_TYPE) {
                $$.type = FLOAT_TYPE;
                if (strcmp($2,"+")==0) {
                        addLine("fadd");
                } else if (strcmp($2,"-")==0) {
                        addLine("fsub");
                }
        } else {
                $$.type = INT_TYPE;
                if (strcmp($2,"+")==0) {
                        addLine("iadd");
                } else if (strcmp($2,"-")==0) {
                        addLine("isub");
                }
        }
};
```

 # SIMPLE_EXPRESSION = TERM | SIGN TERM | SIMPLE_EXPRESSION ‘addop’ TERM

1- sets the type of RHS to the type of the term.

2- similarly it sets the type of RHS to the type of the term and if the sign is negative, it checks whether the term is int or float and adds a new line to the java byte code correspondingly.

3- sets the type of RHS to either float or int and adds a line with the arithmetic operation corresponding to the type of the instruction, it can be added or subtracted to either int or float.

```
term:
        factor {
                $$.type = $1.type;
        }
| term mulop factor {
        if ($1.type == FLOAT_TYPE || $3.type == FLOAT_TYPE) {
                $$.type = FLOAT_TYPE;
                if (strcmp($2,"*")==0) {
                        addLine("fmul");
                } else if (strcmp($2,"/")==0) {
                        addLine("fdiv");
                }
        } else {
                $$.type = INT_TYPE;
                if (strcmp($2,"*")==0) {
                        addLine("imul");
                } else if (strcmp($2,"/")==0) {
                        addLine("idiv");
                }
        }

}
```

 # TERM = FACTOR | TERM ‘mulop’ FACTOR

1- sets the type of RHS to the type of the factor.

2- sets the type of RHS to either float or int and adds a line with the arithmetic operation corresponding to the type of the instruction, it can be multiply or divide to either int or float.

```
factor:
        identifier {
                string id_str($1);
                if (is_valid_id(id_str)) {
                        if (symbol_table[id_str].second == INT_TYPE) {
                                $$.type = INT_TYPE;
                                addLine("iload " + to_string(symbol_table[id_str].first));
                        } else {
                                $$.type = FLOAT_TYPE;
                                addLine("fload " + to_string(symbol_table[id_str].first));
                        }
                } else {
                        string error = id_str + " isn't declared in this scope";
            yyerror(error.c_str());
                }
        }
        | int_value
        {
                $$.type = INT_TYPE;
                addLine("ldc " + to_string($1));
        }
        | float_value
        {
                $$.type = FLOAT_TYPE;
                addLine("ldc " + to_string($1));
        }
        | '(' expression ')' {$$.type = $2.type;};
```

 # FACTOR = ‘id’ | ‘num’ | ‘(‘ EXPRESSION ‘)’

1- In case of id, it is checked that it is already identified and the type of RHS is set to the type of the id. A java bytecode instruction is added to load the id in the stack.

2- In case of num, the lex returns either float_type for a float and int_type for an integer. The type of RHS is set accordingly and an instruction is added to load the constant value in the stack.

3- In case of an expression, the type of RHS is set to the type of the expression.

```
M: {
$$ = javaByteCode.size();
};

sign: addop;
%%
```

 # SIGN = ‘+’ | ‘-’

```
int main(void)
{
  FILE *f;
  f = fopen("input.txt", "r");
  yyin = f;
  yyparse();
  print_output();
}
```

Int main(void): reads input code from input.txt file and parses it then prints the generated java bytecode onto the javaByteCode.j file.

Lex file

It detects the tokens recognized by our java bytecode generator to pass it to the bison after setting values of booleans, id, operators, float and int.

```
%%
"boolean"                               {return boolean_word;}
"int"                                   {return int_word;}
"float"                                 {return float_word;}
"if"                                    {return if_term;}
"else"                                  {return else_term;}
"while"                                                          {return while_token;}
"true"|"false"                          {if(!strcmp(yytext,"true")){
                                            yylval.boolean_val = 1;
                                          } else {
                                            yylval.boolean_val = 0;
                                          }
                                          return boolean;
                                         }

[a-zA-Z][a-zA-z|0-9]*                   {yylval.id_val = strdup(yytext);
                                         return identifier;}

"-"?{num}                               {yylval.int_val = atoi(yytext);
                                         return int_value;}

"-"?{num}.{num}("E"{num})?              {yylval.float_val = atof(yytext);
                                         return float_value;}

"=="|"!="|">="|"<="|"^"|"<"|">"         {yylval.operation = strdup(yytext);
                                         return relop;}

"&&"|"||"|"!"                           {yylval.operation = strdup(yytext);
                                         return boolean_op;}

"*"|"/"                                 {yylval.operation = strdup(yytext);
                                         return mulop;}

"+"|"-"                                 {yylval.operation = strdup(yytext);
                                         return addop;}

";"                                     {return semi_colon;}
"="                                     {return equals;}
"("                                     {return round_open;}
")"                                     {return round_close;}
"{"                                     {return curly_open;}
"}"                                     {return curly_close;}

%%
```

Sample Runs

Rules file

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

Input file

```
if sum, count, pass;
float f;
boolean b;
while (pass != 10)
{
    pass++;
}
if (b) {
    f = 3.14;
}
```

Minimized transition table

state	' '	'!'	'"'	'#'	'$'	'%'	'&'	'''	'('	')'	'*'	'+'	','	'-'	'.'	'/'	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	':'	';'	'<'	'='	'>'	'?'	'@'	'A'	'B'	'C'	'D'	'E'	'F'	'G'	'H'	'I'	'J'	'K'	'L'	'M'	'N'	'O'	'P'	'Q'	'R'
0	-1	1	2	3	-1	-1	-1	-1	4	5	6	7	8	9	10	11	12	12	12	12	12	12	12	12	12	12	13	14	15	16	17	-1	18	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	31	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	32	-1	12	12	12	12	12	12	12	12	12	12	12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*14	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*15	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	33	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*16	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	34	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*17	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	35	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*19	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*20	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*21	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*23	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*24	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*25	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*26	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*27	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*28	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*29	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*30	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*31	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
32	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	45	45	45	45	45	45	45	45	45	45	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*33	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*34	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*35	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*36	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*37	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*38	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*39	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*40	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*41	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*42	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*43	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*44	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*45	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	45	45	45	45	45	45	45	45	45	45	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*46	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*47	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*48	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*49	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*50	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
51	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	56	56	56	56	56	56	56	56	56	56	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*52	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*53	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*54	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*55	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*56	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	56	56	56	56	56	56	56	56	56	56	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
*57	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*58	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*59	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*60	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
*61	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	36	36	36	36	36	36	36	36	36	36	-1	-1	-1	-1	-1	-1	-1	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37

'S'	'T'	'U'	'V'	'W'	'X'	'Y'	'Z'	'['	'\'	']'	'^'	'_'	'`'	'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'	'p'	'q'	'r'	's'	't'	'u'	'v'	'w'	'x'	'y'	'z'	'{'	'|'	'}'	'~'
19	19	19	19	19	19	19	19	20	-1	21	-1	-1	-1	22	23	22	22	24	25	22	22	26	22	22	22	22	22	22	22	22	22	22	22	22	22	27	22	22	22	28	29	30	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	39	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	40	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	41	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	42	38	38	38	38	38	38	38	43	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	44	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	46	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	47	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	48	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	49	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	50	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	52	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	53	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	54	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	55	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	57	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	58	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	59	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	60	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	61	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1
37	37	37	37	37	37	37	37	-1	-1	-1	-1	-1	-1	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	-1	-1	-1	-1

Output

```
if --> if
sum --> id
, --> ,
count --> id
, --> ,
pass --> id
; --> ;
float --> float
f --> id
; --> ;
boolean --> boolean
b --> id
; --> ;
while --> while
( --> (
pass --> id
!= --> relop
10 --> num
) --> )
{ --> {
pass --> id
+ --> addop
+ --> addop
; --> ;
} --> }
```

```
if --> if
( --> (
b --> id
) --> )
{ --> {
f --> id
= --> assign
3.14 --> num
; --> ;
} --> }
```

CFG file:

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

Input code file:

```
int x;
x = 5;
if (x > 2)
{
 x = 0;
}
else
{
 x = 1;
}
```

Parser table file:

-	;	addop	assign	else
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
error	error	error	error	error
SIMPLE_EXPRESSIONEXPRESSION*	synch	error	error	error
SIGNTERMSIMPLE_EXPRESSION~	synch	error	error	error
error	synch	synch	error	error
error	synch	synch	error	error
-	error	error	error	error
error	error	error	error	error
error	epsilon	addopTERMSIMPLE_EXPRESSION~	error	error
error	epsilon	epsilon	error	error
error	epsilon	error	error	error

float	id	if	int
STATEMENT_LIST	STATEMENT_LIST	STATEMENT_LIST	STATEMENT_LIST
STATEMENTSTATEMENT_LIST~	STATEMENTSTATEMENT_LIST~	STATEMENTSTATEMENT_LIST~	STATEMENTSTATEMENT_LIST~
DECLARATION	ASSIGNMENT	IF	DECLARATION
PRIMITIVE_TYPEid;	synch	synch	PRIMITIVE_TYPEid;
float	synch	error	int
synch	synch	if(EXPRESSION){STATEMENT}else{STATEMENT}	synch
synch	synch	synch	synch
synch	idassignEXPRESSION;	synch	synch
error	SIMPLE_EXPRESSIONEXPRESSION*	error	error
error	TERMSIMPLE_EXPRESSION~	error	error
error	FACTORTERM~	error	error
error	id	error	error
error	synch	error	error
PRIMITIVE_TYPEid;STATEMENT_LIST~	ASSIGNMENTSTATEMENT_LIST~	if(EXPRESSION){STATEMENT}else{STATEMENT}STATEMENT_LIST~	PRIMITIVE_TYPEid;STATEMENT_LIST~
error	error	error	error
error	error	error	error
error	error	error	error

	$	(	)	+
METHOD_BODY :	synch	error	error	error
STATEMENT_LIST :	synch	error	error	error
STATEMENT :	synch	error	error	error
DECLARATION :	synch	error	error	error
PRIMITIVE_TYPE :	error	error	error	error
IF :	synch	error	error	error
WHILE :	synch	error	error	error
ASSIGNMENT :	synch	error	error	error
EXPRESSION :	error	SIMPLE_EXPRESSIONEXPRESSION*	synch	SIMPLE_EXPRESSIONEXPRESSION*
SIMPLE_EXPRESSION :	error	TERMSIMPLE_EXPRESSION~	synch	SIGNTERMSIMPLE_EXPRESSION~
TERM :	error	FACTORTERM~	synch	error
FACTOR :	error	(EXPRESSION)	synch	error
SIGN :	error	synch	error	+
STATEMENT_LIST~ :	epsilon	error	error	error
SIMPLE_EXPRESSION~ :	error	error	epsilon	error
TERM~ :	error	error	epsilon	error
EXPRESSION* :	error	error	epsilon	error

mulop	num	relop	while
error	error	error	STATEMENT_LIST
error	error	error	STATEMENTSTATEMENT_LIST~
error	error	error	WHILE
error	error	error	synch
error	error	error	error
error	error	error	synch
error	error	error	while(EXPRESSION){STATEMENT}
error	error	error	synch
error	SIMPLE_EXPRESSIONEXPRESSION*	error	error
error	TERMSIMPLE_EXPRESSION~	synch	error
error	FACTORTERM~	synch	error
synch	num	synch	error
error	synch	error	error
error	error	error	while(EXPRESSION){STATEMENT}STATEMENT_LIST~
error	error	epsilon	error
mulopFACTORTERM~	error	epsilon	error
error	error	relopSIMPLE_EXPRESSION	error

{	}
error	error
error	error
error	synch
error	synch
error	error
error	synch
error	synch
error	synch
error	error
error	error
error	error
error	error
error	error
error	error
error	error
error	error
error	error

Parser output file:

```
METHOD_BODY --> STATEMENT_LIST
STATEMENT_LIST --> STATEMENT STATEMENT_LIST~
STATEMENT --> DECLARATION
DECLARATION --> PRIMITIVE_TYPE id ;
PRIMITIVE_TYPE --> int
match 'int'
match 'id'
match ';'
STATEMENT_LIST~ --> ASSIGNMENT STATEMENT_LIST~
ASSIGNMENT --> id assign EXPRESSION ;
match 'id'
match 'assign'
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION*
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> num
match 'num'
TERM~ --> epsilon
SIMPLE_EXPRESSION~ --> epsilon
EXPRESSION* --> epsilon
match ';'
STATEMENT_LIST~ --> if ( EXPRESSION ) { STATEMENT } else { STATEMENT } STATEMENT_LIST~
match 'if'
match '('
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION*
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> id
```

```
match 'id'
TERM~ --> epsilon
SIMPLE_EXPRESSION~ --> epsilon
EXPRESSION* --> relop SIMPLE_EXPRESSION
match 'relop'
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> num
match 'num'
TERM~ --> epsilon
SIMPLE_EXPRESSION~ --> epsilon
match ')'
match '{'
STATEMENT --> ASSIGNMENT
ASSIGNMENT --> id assign EXPRESSION ;
match 'id'
match 'assign'
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION*
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> num
match 'num'
TERM~ --> epsilon
SIMPLE_EXPRESSION~ --> epsilon
EXPRESSION* --> epsilon
```

```
match ';'
match '}'
match 'else'
match '{'
STATEMENT --> ASSIGNMENT
ASSIGNMENT --> id assign EXPRESSION ;
match 'id'
match 'assign'
EXPRESSION --> SIMPLE_EXPRESSION EXPRESSION*
SIMPLE_EXPRESSION --> TERM SIMPLE_EXPRESSION~
TERM --> FACTOR TERM~
FACTOR --> num
match 'num'
TERM~ --> epsilon
SIMPLE_EXPRESSION~ --> epsilon
EXPRESSION* --> epsilon
match ';'
match '}'
STATEMENT_LIST~ --> epsilon
match '$'
parsing done successfully
```

Input program for code generation

```
int count;
int pass;
pass = 1;
count = pass * 2;
while( pass != 1 ){
        pass = pass + 1;
}
count = pass * 2 + 5;
```

Java Byte Code

```
1 iconst_0
2 istore 1
3 iconst_0
4 istore 2
5 ldc 1
6 istore 2
7 iload 2
8 ldc 2
9 imul
10 istore 1
11 iload 2
12 ldc 1
13 if_icmpeq 15
14 goto 20
15 iload 2
16 ldc 1
17 iadd
18 istore 2
19 goto 11
20 iload 2
21 ldc 2
22 imul
23 ldc 5
24 iadd
25 istore 1
```

Conclusion

The developed project has three parts. Firstly, we developed a lexical analyzer generator that converts the lexical rules given as regular expressions to a minimal DFA that recognizes expected token tokens. Secondly, we developed a parser generator tool that constructs a parsing table given a set of productions for a specified context-free grammar. Lastly, GNU Bison tool was used to generate a java bytecode given a java program, where we annotated a minimized set of java grammar with semantic rules that generates the desired java bytecode in the standard instructions format.

Assumptions

Lexical Rules:

- Spaces represent concatenating operation, even before parentheses
- No spaces precede closure symbols (*, +).
- Range of characters is assumed to be written without spaces (ex: 0-9).
- Regular definitions are case sensitive.

Lexical Analyzer Input:

- Spaces, if exist, are considered as end of token.
- New line is neglected.

NFA:

- Empty transitions (epsilon transitions) are represented by the space character

Parser

- Panic mode recovery is used.
- After the whole vector of tokens is returned from the lexical analyzer it returns a token of the '$' symbol such that the parser can detect that tokens are finished.

Input file

- Each line starts with # and represents the productions of a non terminal.
- Terminals are surrounded by single quotes ex: 'int'.
- Productions are separated by '|' character.
- Each symbol in the production is separated by a space.
- Grammar can be converted to LL(1) grammar by removing left recursion and applying left factoring.

Java Byte Code

We implemented the following:

- Primitive types
- Boolean Expressions
- Arithmetic Expressions
- Assignment statements
- if-else statements
- while loops

Role of each student

Abanoub Ashraf Ezzat Zaki ID: 1

- Minimization of the DFA and creation of the minimized transition table.
- Creation of the lexical analyzer program, which simulates the resulting DFA machine using the maximal munch rule and back tracking error recovery routine.
- Creation of the parser that produces some representation of the leftmost derivation for a correct input, If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing
- Writing the semantic rules for primitive type, declaration and assignment.

Bassam Rageh Ibrahim ID: 15

- Convert NFA to DFA.
- Lex automatic lexical analyser generator (phase 1).
- Building parsing table.
- Lex and Bison files and connecting Bison to Lex.

Eman Rafik ID: 13

- Parsing the input file of the lexical analyzer generator to interpret the regular expressions into its raw expressions.
- Computing first and follow sets of non terminals of the input grammar of the parser generator.
- Writing the semantic rules for arithmetic expressions and while statement.

Reham Mohamed Naguib ID: 19

- Construction of NFA machines for the given regular expressions and combine them together with a new starting state.
- Eliminating grammar left recursion and performing left factoring before generating the parser.
- Writing the semantic rules for boolean expressions and If-else statements to generate java byte code.

References

Literature Review

- https://en.wikipedia.org/wiki/Compiler-compiler#Examples
- https://javacc.github.io/javacc/
- https://www.quora.com/What-does-ANTLR-do
- https://tomassetti.me/antlr-mega-tutorial/
- https://www.gnu.org/software/bison/
- https://en.wikipedia.org/wiki/GNU_Bison

Java byte code

- https://www.javaworld.com/article/2077233/bytecode-basics.html
- https://piazza.com/class_profile/get_resource/k6ng0a2ua975y3/k9o71vdy8clpk?fbclid=IwAR0Wjr-9xNgkc13X_sMkn87Zb2mn4iMlcZA1fd7yE2CHdK0aAMxJ_evhrb0
- https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings

Phase 1 & 2

- https://drive.google.com/file/d/1Qi98OweYwTa6JooOChzACj1q83NeUx6i/view?usp=sharing

Bison and lex

- https://www.youtube.com/watch?v=54bo1qaHAfk&list=PLkB3phqR3X43IRqPT0t1iBfmT5bvn198Z&index=2&t=0s