Parser
Phase 2

Team members:

- Ahmed Mohamed Ali Hassan Aboemera (#7)
- Omar Mohamed Aly Abd Elsalam Barakat (#42)
- Omar Youssry Mahmoud Attia (#44)
- Karim Waguih El Azzouni (#49)

Contents:

1- Project Description
2- Project Design
    - Parsing
    - eliminating left recursion and left factoring the grammar
    - Calculation of FIRST and FOLLOW
    - Building Predictive table
    - Receiving tokens from lexical analyzer
    - Simulating the code
3- About the grammar
4- Program Flow
5- Code Snippets
6- Algorithms and Data structures

1- Project Description:

Your task in this phase of the assignment is to design and implement an LL(1) parser generator tool.
The parser generator expects an LL (1) grammar as input. It should compute First and Follow sets and uses them to construct a predictive parsing table for the grammar.
The table is to be used to drive a predictive top-down parser. If the input grammar is not LL (1), an appropriate error message should be produced.
The generated parser is required to produce some representation of the leftmost derivation for a correct input. If an error is encountered, a panic-mode error recovery routine is to be called to print an error message and to resume parsing.
The parser generator is required to be tested using the given context free grammar of a small subset of Java. Of course, you have to modify the grammar to allow predictive parsing.
Combine the lexical analyzer generated in phase 1 and parser such that the lexical analyzer is to be called by the parser to find the next token. Use the simple program given in phase 1 to test the combined lexical analyzer and parser.

Bonus Task
Automatically eliminating grammar left recursion and performing left factoring before generating the parser will be considered a bonus work.

2- Project Design:

- **Parsing:**
  At the very beginning , method **modify_grammer_file** is invoked to concatenate the production rules defined at multiple lines. The return of this method is a **vector<string>** each entry in this vector defines a production rule.
  Then an instance of **initiator** class is made to initiate the data structures used afterwards in the project.
  The data structures are the following:
  - **vector<string> terminals** :: contains all terminals of the grammar.
  - **vector<string> non_terminals** :: contains the names of all non_terminals in the grammar.
  - **vector<vector<vector<string>>> non_terminals_defs** :: each entry in this vector defines the **production rule** of the corresponding non_terminal in non_terminals vector.
    The vector<vector<string>> defines the production rule of a non_terminal as a SOP “Sum of products”

- **Eliminating left recursion and left factoring the grammar:**

First Elimination of Left recursion is performed according to the given algorithm in the lecture.

- Arrange non-terminals in some order: $A_1 \dots A_n$
- **for** i **from** 1 **to** n **do** {
  - **for** j **from** 1 **to** i-1 **do** {
    replace each production
    $A_i \rightarrow A_j \gamma$
    by
    $A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$
    where $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$
  }
  - eliminate immediate left-recursions among A_i productions
}

Afterwards Left Elimination is performed according to the rules in the lecture

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \dots \mid \alpha\beta_n \mid \gamma_1 \mid \dots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \dots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \dots \mid \beta_n$$

- **Calculation of FIRST and FOLLOW:**

First and follow are calculated by functions provided in class Utilities.
The first is computed by the function “get_first(string non_terminal)” which returns all terminals in first of the passed non_terminal.

The follows are computed by the function “compute_follow()” which returns all terminals in first of the passed non_terminal. This function uses the two functions “compute_follow_rule1()” which applies rule 1 to the computed follows and “compute_follow_rule2()” which applies rule 2 to the computed follows.

The computed firsts and follows are stored in maps to be easily retrieved later.

- **Building Predictive Table:**
  Predictive table is built using the FIRST and FOLLOW functions for each non-terminal calculated earlier, by combining algorithm 4.31 in the textbook and the panic mode error recovery method in LL(1) parser.
  For the production A = α is required to be input at the input symbol a, and a production A = β already exists, an exception is thrown to indicate that the grammar is not LL(1) and the while parsing operation is aborted.

- **Receiving tokens from the lexical analyzer:**
  (Disclaimer: Since our lexical analyzer from phase 1 was buggy, we decided that the input to the parser would be a text file of the tokens that should be received by a correct lexical analyzer)
  Input buffer of the parser is filled with tokens from the lexical analyzer by the function “getNextToken()”.

- **Simulating the code:**
  Code is simulated using algorithm 4.34 in the textbook, which is a nonrecursive predictive parsing technique that uses the predictive table and an explicit stack to parse the input tokens. The output is written into output.txt which is a text file that contains a step-by-step left-most derivation of the source code inputted to the lexical analyzer.

4- About the grammar:

Original Grammar

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

Grammar after eliminating left recursion

```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT STATEMENT_LIST`
# STATEMENT_LIST` = STATEMENT STATEMENT_LIST` | \L
# 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 SIMPLE_EXPRESSION` | SIGN TERM SIMPLE_EXPRESSION`
# SIMPLE_EXPRESSION` = ‘addop’ TERM SIMPLE_EXPRESSION` | \L
# TERM = FACTOR TERM`
# TERM` = ‘mulop’ FACTOR TERM` | \L
# FACTOR = ‘id’ | ‘num’ | ‘(‘ EXPRESSION ‘)’
# SIGN = ‘+’ | ‘-‘
```

Grammar after left factoring

```
# METHOD_BODY = STATEMENT_LIST
# STATEMENT_LIST = STATEMENT STATEMENT_LIST`
# STATEMENT_LIST` = STATEMENT STATEMENT_LIST` | \L
# 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*
# SIMPLE_EXPRESSION* = \L | 'relop' SIMPLE_EXPRESSION
# SIMPLE_EXPRESSION = TERM SIMPLE_EXPRESSION` | SIGN TERM SIMPLE_EXPRESSION`
# SIMPLE_EXPRESSION` = ‘addop’ TERM SIMPLE_EXPRESSION` | \L
# TERM = FACTOR TERM`
# TERM` = ‘mulop’ FACTOR TERM` | \L
# FACTOR = ‘id’ | ‘num’ | ‘(‘ EXPRESSION ‘)’
# SIGN = ‘+’ | ‘-‘
```

FIRST

```
FIRST(METHOD_BODY):            {'int', 'float', 'if', 'while', 'id'}
FIRST(STATEMENT_LIST):         {'int', 'float', 'if', 'while', 'id'}
FIRST(STATEMENT_LIST`):        {'int', 'float', 'if', 'while', 'id', '\L'}
FIRST(STATEMENT):              {'int', 'float', 'if', 'while', 'id'}
FIRST(DECLARATION):            {'int', 'float'}
FIRST(IF):                     {'if'}
FIRST(WHILE):                  {'while'}
FIRST(ASSIGNMENT):             {'id'}
FIRST(PRIMITIVE_TYPE):         {'int', 'float'}
FIRST(EXPRESSION):             {'id', 'num', '(', '+', '-'}
FIRST(SIMPLE_EXPRESSION*):     {'\L', 'relop'}
FIRST(SIMPLE_EXPRESSION):      {'id', 'num', '(', '+', '-'}
FIRST(SIMPLE_EXPRESSION`):     {'addop', '\L'}
FIRST(TERM):                   {'id', 'num', '('}
FIRST(TERM`):                  {'mulop', '\L'}
FIRST(SIGN):                   {'+', '-'}
FIRST(FACTOR):                 {'id', 'num', '('}
```

FOLLOW

```
FOLLOW(METHOD_BODY)                 : {'$'}
FOLLOW(STATEMENT_LIST)              : {'$'}
FOLLOW(STATEMENT_LIST`)             : {'$'}
FOLLOW(STATEMENT)                   : {'int', 'float', 'if', 'while', 'id', '$', '}'}
FOLLOW(DECLARATION)                 : {'int', 'float', 'if', 'while', 'id', '$', '}'}
FOLLOW(IF)                          : {'int', 'float', 'if', 'while', 'id', '$', '}'}
FOLLOW(WHILE)                       : {'int', 'float', 'if', 'while', 'id', '$', '}'}
FOLLOW(ASSIGNMENT)                  : {'int', 'float', 'if', 'while', 'id', '$', '}'}
FOLLOW(PRIMITIVE_TYPE)              : {'id'}
FOLLOW(EXPRESSION)                  : {')', ';'}
FOLLOW(SIMPLE_EXPRESSION)           : {')', ';', 'relop'}
FOLLOW(SIMPLE_EXPRESSION*)          : {')', ';'}
FOLLOW(SIMPLE_EXPRESSION`)          : {')', ';', 'relop'}
FOLLOW(TERM)                        : {'addop',')', ';', 'relop'}
FOLLOW(TERM`)                       : {'addop',')', ';', 'relop'}
FOLLOW(SIGN)                        : {'id', 'num', '('}
FOLLOW(FACTOR)                      : {'mulop','addop',')', ';', 'relop'}
```

Sample code:

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

Tokenized code:

```
int
id
;
id
assign
num
;
if
(
id
relop
num
)
{
id
assign
num
;
}
```

Predictive Parsing Table (after enabling panic mode)

```
ASSIGNMENT:
        $:
                synch
        float:
                synch
        id:
                ‘id’ ‘assign’ EXPRESSION ‘;’
        if:
                synch
        int:
                synch
        while:
                synch
        }:
                synch
DECLARATION:
        $:
                synch
        float:
                PRIMITIVE_TYPE ‘id’ ‘;’
        id:
                synch
        if:
                synch
        int:
                PRIMITIVE_TYPE ‘id’ ‘;’
        while:
                synch
        }:
                synch
EXPRESSION:
        (:
                SIMPLE_EXPRESSION EXPRESSION*
        ):
                synch
        +:
                SIMPLE_EXPRESSION EXPRESSION*
        -:
                SIMPLE_EXPRESSION EXPRESSION*
        ;:
                synch
        id:
                SIMPLE_EXPRESSION EXPRESSION*
        num:
                SIMPLE_EXPRESSION EXPRESSION*
EXPRESSION*:
        ):
                \L
        ;:
                \L
        relop:
                ‘relop’ SIMPLE_EXPRESSION
FACTOR:
        (:
                ‘(‘ EXPRESSION ‘)’
        ):
                synch
        ;:
                synch
        addop:
                synch
        id:
                ‘id’
        mulop:
                synch
        num:
                ‘num’
        relop:
                synch
IF:
        $:
                synch
        float:
                synch
        id:
                synch
```

```
            if:
                        ‘if’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’
            int:
                        synch
            while:
                        synch
            }:
                        synch
METHOD_BODY:
            $:
                        synch
            float:
                        STATEMENT_LIST
            id:
                        STATEMENT_LIST
            if:
                        STATEMENT_LIST
            int:
                        STATEMENT_LIST
            while:
                        STATEMENT_LIST
PRIMITIVE_TYPE:
            float:
                        ‘float’
            id:
                        synch
            int:
                        ‘int’
SIGN:
            (:
                        synch
            +:
                        ‘+’
            -:
                        ‘-‘
            id:
                        synch
            num:
                        synch
SIMPLE_EXPRESSION:
            (:
                        TERM SIMPLE_EXPRESSION'
            ):
                        synch
            +:
                        SIGN TERM SIMPLE_EXPRESSION'
            -:
                        SIGN TERM SIMPLE_EXPRESSION'
            ;:
                        synch
            id:
                        TERM SIMPLE_EXPRESSION'
            num:
                        TERM SIMPLE_EXPRESSION'
            relop:
                        synch
SIMPLE_EXPRESSION':
            ):
                        \L
            ;:
                        \L
            addop:
                        ‘addop’ TERM SIMPLE_EXPRESSION'
            relop:
                        \L
STATEMENT:
            $:
                        synch
            float:
                        DECLARATION
            id:
                        ASSIGNMENT
            if:
                        IF
            int:
                        DECLARATION
            while:
                        WHILE
```

```
          }:
                    synch
STATEMENT_LIST:
          $:
                    synch
          float:
                    STATEMENT STATEMENT_LIST'
          id:
                    STATEMENT STATEMENT_LIST'
          if:
                    STATEMENT STATEMENT_LIST'
          int:
                    STATEMENT STATEMENT_LIST'
          while:
                    STATEMENT STATEMENT_LIST'
STATEMENT_LIST':
          $:
                    \L
          float:
                    STATEMENT STATEMENT_LIST'
          id:
                    STATEMENT STATEMENT_LIST'
          if:
                    STATEMENT STATEMENT_LIST'
          int:
                    STATEMENT STATEMENT_LIST'
          while:
                    STATEMENT STATEMENT_LIST'
TERM:
          (:
                    FACTOR TERM'
          ):
                    synch
          ;:
                    synch
          addop:
                    synch
          id:
                    FACTOR TERM'
          num:
                    FACTOR TERM'
          relop:
                    synch
TERM':
          ):
                    \L
          ;:
                    \L
          addop:
                    \L
          mulop:
                    ‘mulop’ FACTOR TERM'
          relop:
                    \L
WHILE:
          $:
                    synch
          float:
                    synch
          id:
                    synch
          if:
                    synch
          int:
                    synch
          while:
                    ‘while’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATEMENT ‘}’
          }:
                    synch
```

5- Project flow:

The input to the program is the grammar file, not necessarily left factored or with left recursion eliminated.
The first step is to put the grammar into LL(1) format by eliminating the left recursion and left factoring. Then, the first and follow of each non-temrinal is calculated following the given algorithms in the lecture and in the reference.

Using the first and follow, we can now construct the parsing table.
After that, the tokenized java code can be parsed using a stack and the parsing table, producing a left derivation of the given code. Parser uses the Panic Mode recovery technique when errors occur to skip unmatched tokens.

6- Code Snippets:

Eliminator.h

```
/*
 * Eliminator.h
 *
 *  Created on: 13 Apr 2015
 *      Author: ahmedaboemera
 */

#ifndef ELIMINATOR_H_
#define ELIMINATOR_H_

#include "Initiator.h"
using namespace std;

class Initiator;

class Eliminator{
public:
    Initiator* initiator;
    Eliminator(Initiator* i);
    void eliminate_LF() ;
    void eliminate_LR();
    void print_grammer();
private:
    bool find (vector<string>* taken, string s);
    vector<vector<int> > get_left_factors(int index);
    void eliminate_LF(int index) ;
    bool probe_for_lf(int index);
    int find_index(string s);
    vector<vector<string > > replace(int i);
    void eliminate_from_updated(vector<vector<string> > replaced, int index);
};


#endif /* ELIMINATOR_H_ */
```

Initiator.h

```
/*
 * Parser.h
 *  Created on: 12 Apr 2015
 *      Author: ahmedaboemera
 */

#ifndef INITIATOR_H_
#define INITIATOR_H_

#include <vector>
#include <string>
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <map>
#include "Utilities.h"

using namespace std;

class Utilities;

class Initiator {
public:
    Initiator();
    string starting;
    vector<string>* terminals;
    vector<string>* non_terminals;
    vector<vector<vector<string> > >* non_terminal_defs;
    void get_terminals_and_nonterminals(vector<string>* lines);
    bool is_terminal(string s);
    bool is_non_terminal(string s);
    string get_starting();
//  map<string,map<string, vector<string> > > built_predictive_table();
    vector<vector<string> > get_def(string s);
    string trim(string x);
private:
//  Utilities util;
    void fill_map(vector<string>* lines);
    void finalize(vector<vector<string> > non_t_defs);
    vector<string> split(string s, int start, char regex);
    int get_eq(string s);
    vector<string>* get_terminals(string s);
    string get_nonterminal(string s);
};
#endif /* INITIATOR_H_ */
```

Parser.h

```
/*
 * Parser.h
 *
 *  Created on: 17 Apr 2015
 *      Author: ahmedaboemera
 */

#ifndef SRC_PARSER_H_
#define SRC_PARSER_H_

#include "Utilities.h"
#include "Initiator.h"

class Parser{
public:
    Parser(Initiator* i, Utilities* u);
    void built_predictive_table();
    vector<string> get_entry(string non_terminal, string terminal);
    void print_predictive_table();
    string remove_uni_quotes(string s);
private:
    map<string,map<string, vector<string> > > predictive_table;
    Initiator* i;
    Utilities* u;
};


#endif /* SRC_PARSER_H_ */
```

Utilities.h

```

#ifndef UTILITIES_H_
#define UTILITIES_H_

#include <algorithm>
#include <vector>
#include <map>
#include <set>

#include "Initiator.h"

using namespace std;

class Initiator;

class Utilities{
    public:
        Utilities(Initiator* i);
        map<string, vector<string> > compute_first();
        void compute_follow();
        vector<set<string> > get_first(string non_terminal);
        set<string> get_follow(string non_terminal);
        void print_set(set<string> firsts);
        void print_first();
        void print_follow();

    private:
        Initiator *initiator;
        const string EPSLON = "\\L";
        map<string, vector<set<string> > > non_terminals_first;
        map<string, set<string> > non_terminals_follow;
        set<string> compute_follow_rule1(string non_terminal);
        void compute_follow_rule2(string non_terminal);
        vector<string>::iterator vector_find(vector<string>* vec, string str);
        set<string> union_sets(vector<set<string> > vec);
};


#endif UTILITIES_H_
```

7- Algorithms and data structures:

- C++ STL's structures; mainly:
    - map
    - vector
    - set
    - stack (for parsing)
- C++ STL's algorithms; mainly:
    - sort
    - insert (to union sets)

Left-most derivation

METHOD_BODY
STATEMENT_LIST
STATEMENT STATEMENT_LIST'
DECLARATION STATEMENT_LIST'
PRIMITIVE_TYPE ‘id’ ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ STATEMENT STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ASSIGNMENT STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ EXPRESSION ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ SIMPLE_EXPRESSION EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ TERM SIMPLE_EXPRESSION' EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ SIMPLE_EXPRESSION' EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ EXPRESSION* ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ STATEMENT STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ IF STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ EXPRESSION ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ SIMPLE_EXPRESSION EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ TERM SIMPLE_EXPRESSION' EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ SIMPLE_EXPRESSION' EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ EXPRESSION* ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ SIMPLE_EXPRESSION ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ TERM SIMPLE_EXPRESSION' ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ FACTOR TERM' SIMPLE_EXPRESSION' ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ TERM' SIMPLE_EXPRESSION' ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ SIMPLE_EXPRESSION' ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ STATEMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ASSIGNMENT ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ EXPRESSION ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ SIMPLE_EXPRESSION EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ TERM SIMPLE_EXPRESSION' EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ FACTOR TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ ‘num’ TERM' SIMPLE_EXPRESSION' EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ ‘num’ SIMPLE_EXPRESSION' EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ ‘num’ EXPRESSION* ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'
‘int’ ‘id’ ‘;’ ‘id’ ‘assign’ ‘num’ ‘;’ ‘if’ ‘(‘ ‘id’ ‘relop’ ‘num’ ‘)’ ‘{‘ ‘id’ ‘assign’ ‘num’ ‘;’ ’}’ ‘else’ ‘{’ STATEMENT ‘}’ STATEMENT_LIST'