Hand-coded lexer and parser. Example.

December 27, 2020

1 Context-free grammar

1.1 Rules

$$
\begin{aligned}
\langle program \rangle &\rightarrow \textbf{int main } '(' \ ')' \ \langle block \rangle \\
\langle block \rangle &\rightarrow '\{' \ \langle content \rangle \\
\langle content \rangle &\rightarrow \langle decl\text{–}list \rangle \ \langle instr\text{–}list \rangle \\
\langle content \rangle &\rightarrow \langle instr\text{–}list \rangle \\
\langle decl\text{–}list \rangle &\rightarrow \langle decl\text{–}list \rangle \ \langle decl \rangle \\
\langle decl\text{–}list \rangle &\rightarrow \epsilon \\
\langle decl \rangle &\rightarrow \langle decl\text{–}type \rangle \ \langle ident\text{–}list \rangle \ ';' \\
\langle decl\text{–}type \rangle &\rightarrow \textbf{int} \mid \textbf{double} \\
\langle ident\text{–}list \rangle &\rightarrow \langle ident\text{–}list \rangle \ ',' \ \textbf{\textit{ident}} \\
\langle ident\text{–}list \rangle &\rightarrow \textbf{\textit{ident}} \\
\langle instr\text{–}list \rangle &\rightarrow \langle instr\text{–}list \rangle \ \langle instr \rangle \\
\langle instr\text{–}list \rangle &\rightarrow \epsilon \\
\langle instr \rangle &\rightarrow \langle instr\text{–}assign \rangle \mid \langle instr\text{–}return \rangle \mid \langle instr\text{–}if \rangle \mid \langle instr\text{–}empty \rangle \\
\langle instr\text{–}assign \rangle &\rightarrow \langle ident \rangle \ '=' \ \langle expr \rangle \ ';' \\
\langle instr\text{–}return \rangle &\rightarrow \textbf{return } \langle expr \rangle \ ';' \\
\langle instr\text{–}empty \rangle &\rightarrow ';' \\
\langle expr \rangle &\rightarrow \langle expr \rangle \ '+' \ \langle term \rangle \\
\langle expr \rangle &\rightarrow \langle term \rangle \\
\langle term \rangle &\rightarrow \langle term \rangle \ '*' \ \langle fact \rangle \\
\langle term \rangle &\rightarrow \langle fact \rangle \\
\langle fact \rangle &\rightarrow '(' \ \langle expr \rangle \ ')' \\
\langle fact \rangle &\rightarrow \textbf{\textit{ident}} \\
\langle fact \rangle &\rightarrow \textbf{\textit{int\text{–}const}} \\
\langle fact \rangle &\rightarrow \textbf{\textit{real\text{–}const}} \\
\langle instr\text{–}if \rangle &\rightarrow \textbf{if } '(' \ \langle cond \rangle \ ')' \ \langle instr\text{–}block \rangle \textbf{ else } \langle instr\text{–}block \rangle \\
\langle instr\text{–}block \rangle &\rightarrow \langle instr \rangle \mid \langle block \rangle \\
\langle cond \rangle &\rightarrow \langle rel\text{–}operand \rangle \ \langle rel\text{–}operator \ \rangle \langle rel\text{–}operand \rangle \\
\langle rel\text{–}operand \rangle &\rightarrow \textbf{\textit{ident}} \mid \textbf{\textit{int\text{–}const}} \mid \textbf{\textit{real\text{–}const}} \\
\langle rel\text{–}operator \rangle &\rightarrow '<' \mid '>' \mid '==' \mid '>=' \mid '<=' \mid '! ='
\end{aligned}
$$

1.2 Terminal symbols

Each number represents a lexical class category.

1. **int**
2. **double**
3. **main**
4. **return**
5. **if**
6. **else**
7. ***ident***
8. ***int–const***
9. ***real–const***
10. $'.'$
11. $','$
12. $';'$
13. $'('$
14. $')'$
15. $'\{'$
16. $'\}'$
17. $'='$
18. $'+'$
19. $'*'$
20. $'<'$
21. $'>'$
22. $'=='$
23. $'>='$

24. $'<='$
25. $'!='$

Remark.

- Names in bold are *keywords*,
- Names in italic and bold are:
  - *identifiers* (***ident***),
  - *numeric constants* (***int–const*** and ***real–const***).
- Other elements are:
  - *separators*: $'.'$, $','$, $';'$, $'('$, $')'$, $'\{'$, $'\}'$,
  - *operators*: $'='$, $'+'$, $'*'$, $'<'$, $'>'$, $'=='$, $'>='$, $'<='$, $'!='$

1.3 Non-terminal symbols

- $\langle program \rangle$
- $\langle block \rangle$
- $\langle content \rangle$
- $\langle decl–list \rangle$
- $\langle decl \rangle$
- $\langle decl–type \rangle$
- $\langle ident–list \rangle$
- $\langle instr–block \rangle$
- $\langle instr \rangle$
- $\langle block \rangle$
- $\langle instr–list \rangle$
- $\langle instr–return \rangle$
- $\langle instr–assign \rangle$
- $\langle instr–empty \rangle$

- $\langle instr\text{–}if \rangle$
- $\langle expr \rangle$
- $\langle term \rangle$
- $\langle fact \rangle$
- $\langle cond \rangle$
- $\langle rel\text{–}operand \rangle$
- $\langle rel\text{–}operator \rangle$

1.4 Input file example

```
int main() {
  int n, m;
  double x, y;
  n = 3;
  m = n + 4;
  x = 2.25;
  y = x * 2;
  if (m < y)
    m = n + 10;
  else {
    y = y * .5;
    x = x + 5.;
  }
  return 0;
}
```

2 The scanner

2.1 Regular definitions

$$
\begin{aligned}
\textbf{\textit{ident}} &\rightarrow [a-zA-Z][a-zA-Z0-9]^* \\
\textbf{\textit{int–const}} &\rightarrow 0 \\
\textbf{\textit{int–const}} &\rightarrow [1-9][0-9]^* \\
\textbf{\textit{real–const}} &\rightarrow \textbf{\textit{int–const}}\ '.'\ \textbf{\textit{int–const}} \\
\textbf{\textit{real–const}} &\rightarrow \textbf{\textit{int–const}}\ '.' \\
\textbf{\textit{real–const}} &\rightarrow '.'\ \textbf{\textit{int–const}}
\end{aligned}
$$

2.2 Finite automata used for lexical analysis

The finite automaton for the *identifier* lexical category is presented in Figure 1.

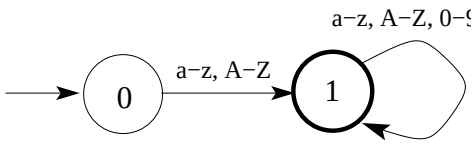

Figure 1: Finite automaton for the *identifier* lexical category

The finite automaton for the *integer − constant* lexical category is presented in Figure 2.

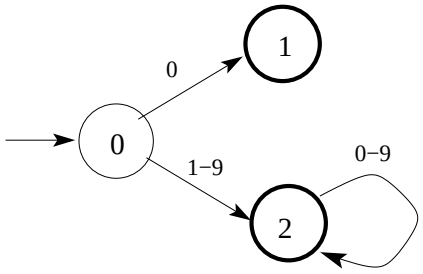

Figure 2: Finite automaton for the *integer − constant* lexical category (denoted by MCI)

The non-deterministic finite automaton for the *real − constant* lexical category is presented in Figure 3.

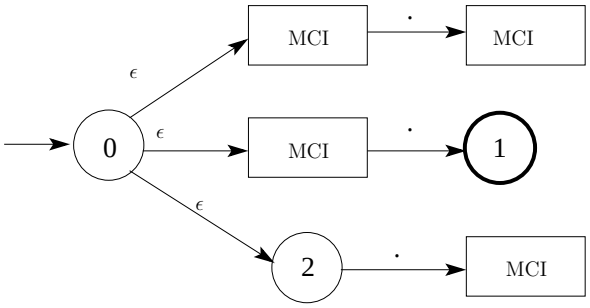

Figure 3: Non-deterministic finite automaton for the $real-constant$ lexical category

The deterministic finite automaton for both $real-constant$ and $integer-constant$ lexical categories is presented in Figure 4.

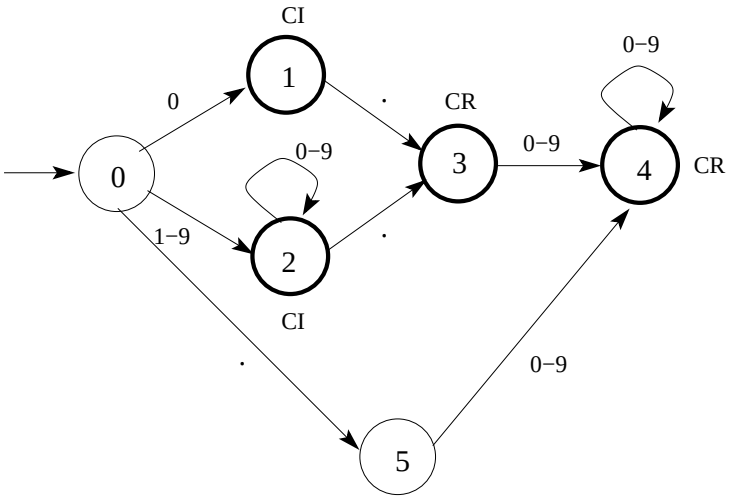

Figure 4: Deterministic finite automaton for $real-constant$ and $integer-constant$ lexical categories

Finally, in Figure 5 is presented a deterministic finite automaton for three lexical categories: $ident$, $real-constant$ and $integer-constant$.

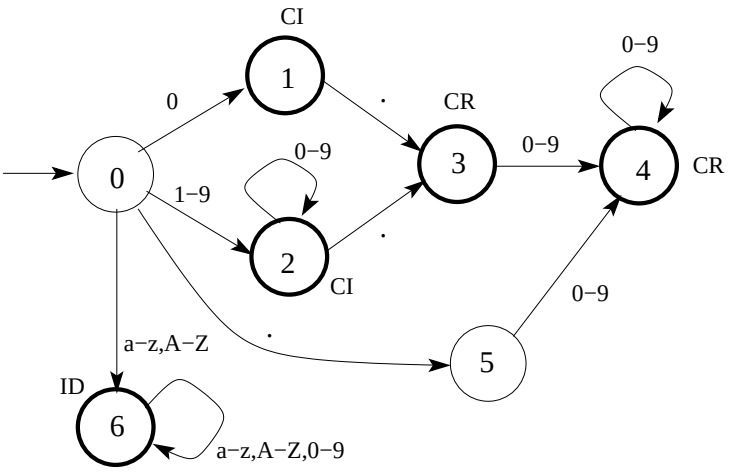

Figure 5: Deterministic finite automaton for *ident*, $real-constant$ and *integer–constant* lexical categories

Finally, in Figure 6 is presented a deterministic finite automaton for all lexical categories.

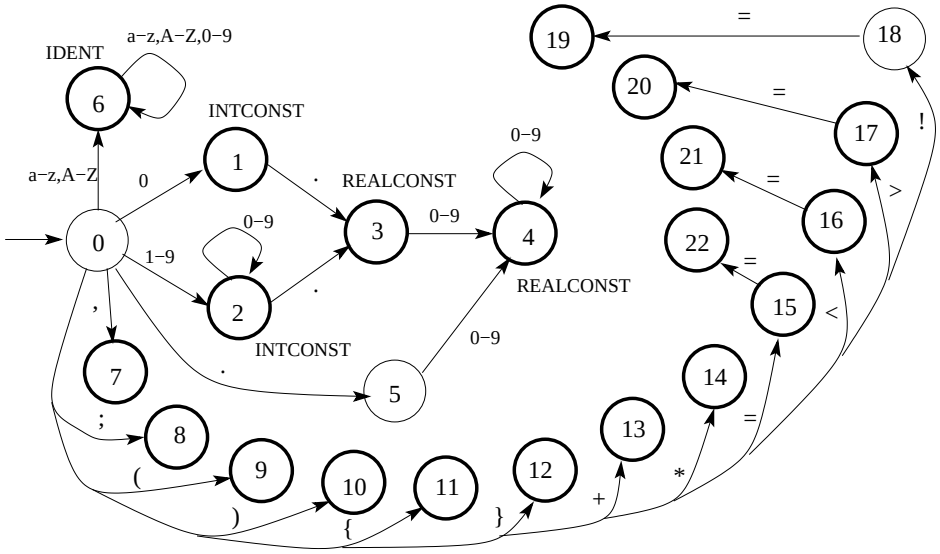

Figure 6: Deterministic finite automaton for all lexical categories

The names of some final states in this DFA are:

- State 7: $COMMA$,
- State 8: $SEMICOLON$,

- State 9: $LPAREN$,
- State 10: $RPAREN$,
- State 11: $LBRACE$,
- State 12: $RBRACE$,
- State 13: $PLUS$,
- State 14: $TIMES$,
- State 15: $ASSIGN$,
- State 16: LT,
- State 17: GT,
- State 19: NEQ,
- State 20: GTE,
- State 21: LTE,
- State 22: EQ.

2.3 Source files

exceptions.h

```
#pragma once

#ifndef EXP
#define EXP

#include <string>

class Exception {
  std::string mess;
public:
  Exception(std::string str): mess(str) {}
  void print();
};

#endif
```

exceptions.cpp

```
#include "exceptions.h"

#include <iostream>

void Exception::print() {
  std::cout << mess;
}
```

toktype.h

```
#pragma once

#ifndef TKTYPE
#define TKTYPE

#include <string>
#include <map>

// enum to store the type of a token
enum class TokenType {
  INTCONST, REALCONST, IDENT, INT, DOUBLE, MAIN, RETURN, IF, ELSE,
  PLUS, TIMES, ASSIGN, LPAREN, RPAREN, LBRACE, RBRACE, COMMA,
  SEMICOLON, EQ, NEQ, LT, LTE, GT, GTE, END
};

// map used to associate the token type of a keyword
typedef std::map<std::string, TokenType> KeywordMap;

// map used to write the type of a token
typedef std::map<TokenType, std::string> TokenTypeMap;

#endif
```

token.h

```
#pragma once

#ifndef TOK
#define TOK

#include <fstream>

#include "toktype.h"
```

```
class Token {
  TokenType type;
  std::string lexeme;
  union {
    int intConst;
    double realConst;
    std::string ident;
    short oth;
  };
public:
  Token(TokenType t, std::string word, int n):
    type(t), lexeme(word), intConst(n) {}
  Token(TokenType t, std::string word, double x) :
    type(t), lexeme(word), realConst(x) {}
  Token(TokenType t, std::string word, std::string id):
    type(t), lexeme(word), ident(id) {}
  Token(TokenType t, std::string word):
    type(t), lexeme(word), oth(1) {}
  // copy constructor is implicit defaulted
  Token(const Token& tok);
  // destructor is implicit defaulted
  ~Token() {}
  // assignment operator is implicit defaulted
  Token& operator=(const Token& tok);
  TokenType getType() const { return type; }
  int getInt() const { return intConst; }
  double getReal() const { return realConst; }
  std::string getIdent() const { return ident; }
  void print(std::ofstream& out) const;
};

#endif
```

token.cpp

```
#include "token.h"

#include <iostream>
#include <string>
#include <map>

// map used to write the type of a token
TokenTypeMap tokenTypeMap{ {TokenType::INTCONST, "INTCONST"},
```

```
  {TokenType::REALCONST, "REALCONST"}, {TokenType::IDENT, "IDENT"},
  {TokenType::INT, "INT"}, {TokenType::DOUBLE, "DOUBLE"},
  {TokenType::MAIN, "MAIN"}, {TokenType::RETURN, "RETURN"},
  {TokenType::IF, "IF"}, {TokenType::ELSE, "ELSE"},
  {TokenType::PLUS, "PLUS"}, {TokenType::TIMES, "TIMES"},
  {TokenType::ASSIGN, "ASSIGN"}, {TokenType::LPAREN, "LPAREN"},
  {TokenType::RPAREN, "RPAREN"}, {TokenType::LBRACE, "LBRACE"},
  {TokenType::RBRACE, "RBRACE"}, {TokenType::COMMA, "COMMA"},
  {TokenType::SEMICOLON, "SEMICOLON"}, {TokenType::EQ, "EQ"},
  {TokenType::NEQ, "NEQ"}, {TokenType::LT, "LT"},
  {TokenType::LTE, "LTE"}, {TokenType::GT, "GT"},
  {TokenType::GTE, "GTE"}, {TokenType::END, "END"}
};

Token::Token(const Token& tok) {
  lexeme = tok.lexeme;
  // copy operations
  switch (tok.type) {
  case TokenType::INTCONST:
    intConst = tok.intConst;
    break;
  case TokenType::REALCONST:
    realConst = tok.realConst;
    break;
  case TokenType::IDENT:
    new (&ident)(std::string)(tok.ident);
    break;
  default:
    oth = tok.oth;
    break;
  }
  type = tok.type;
}

Token& Token::operator=(const Token& tok) {
  lexeme = tok.lexeme;
  if (type == TokenType::IDENT && tok.type == TokenType::IDENT) {
    ident = tok.ident;
    return *this;
  }
  // the current token type = IDENT
  if (type == TokenType::IDENT) {
    ident.~basic_string<char>();  // destroy explicitly
```

```
  }
  // copy operations
  switch (tok.type) {
  case TokenType::INTCONST:
    intConst = tok.intConst;
    break;
  case TokenType::REALCONST:
    realConst = tok.realConst;
    break;
  case TokenType::IDENT:
    new (&ident)(std::string)(tok.ident);
    break;
  default:
    oth = tok.oth;
    break;
  }
  type = tok.type;
  return *this;
}

void Token::print(std::ofstream& out) const {
  out << "Token␣{\n";
  out << "\tLexeme␣=␣" << lexeme << std::endl;
  out << "\tToken␣type␣=␣" << tokenTypeMap[type] << std::endl;
  switch (type) {
  case TokenType::INTCONST:
    out << "\tLValue␣=␣";
    out << intConst << std::endl;
    break;
  case TokenType::REALCONST:
    out << "\tLValue␣=␣";
    out << realConst << std::endl;
    break;
  case TokenType::IDENT:
    out << "\tLValue␣=␣";
    out << ident << std::endl;
    break;
  default:
    break;
  }
  out << "}\n";
}
```

scanner.h

```
#pragma once

#ifndef SCAN
#define SCAN

#include <fstream>
#include <string>
#include <vector>
#include "token.h"

class Scanner {
  std::string fileName;
  std::ifstream inputFile;
  long currPosition;
public:
  Scanner(std::string name);
  Token nextToken();
};

#endif
```

scanner.cpp

```
#include <algorithm>
#include <iostream>
#include <string>

#include "scanner.h"
#include "token.h"
#include "toktype.h"
#include "exceptions.h"


// map used to associate the token type of a keyword
KeywordMap keywordMap{ {"int", TokenType::INT},
  {"double", TokenType::DOUBLE}, {"main", TokenType::MAIN},
  {"return", TokenType::RETURN}, {"if", TokenType::IF},
  {"else", TokenType::ELSE},
};


Scanner::Scanner(std::string name) : fileName(name) {
  inputFile.open(name);
```

```
  if (!inputFile.is_open()) {
    throw Exception("Input␣file␣does␣not␣exist");
  }
  currPosition = inputFile.tellg();
};

Token Scanner::nextToken() {
  char c, c1;
  // read from the current position
  inputFile.seekg(currPosition, std::ios::beg);
  while ((c = inputFile.get()) && inputFile.good()) {
    if ((c == '␣') || (c == '\n') || (c == '\t')) {   // skip spaces
      while ((c = inputFile.get()) && (c == '␣' || c == '\n') || (c == '
    }
    switch (c) {
    case '+':
    {
      // lexeme
      std::string tLexeme;
      tLexeme.push_back(c);
      // token
      Token t(TokenType::PLUS, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    case '*':
    {
      // lexeme
      std::string tLexeme;
      tLexeme.push_back(c);
      // token
      Token t(TokenType::TIMES, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    case ',':
    {
      // lexeme
      std::string tLexeme;
      tLexeme.push_back(c);
      // token
```

```
    Token t(TokenType::COMMA, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case ';':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    // token
    Token t(TokenType::SEMICOLON, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case '(':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    // token
    Token t(TokenType::LPAREN, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case ')':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    // token
    Token t(TokenType::RPAREN, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case '{':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
```

```
    // token
    Token t(TokenType::LBRACE, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case '}':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    // token
    Token t(TokenType::RBRACE, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case '!':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    inputFile >> c1;
    if (c1 == '=') {
      tLexeme.push_back(c1);
      // token
      Token t(TokenType::NEQ, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    else {
      throw Exception("Illegal character after !");
    }
  }
  case '=':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    inputFile >> c1;
    if (c1 == '=') {
      tLexeme.push_back(c1);
```

```
      // token
      Token t(TokenType::EQ, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    else {
      // assignment operator
      // push back the last char in the stream
      inputFile.putback(c1);
      // token
      Token t(TokenType::ASSIGN, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
  }
  case '<':
  {
    // lexeme
    std::string tLexeme;
    tLexeme.push_back(c);
    inputFile >> c1;
    if (c1 == '=') {
      tLexeme.push_back(c1);
      // token
      Token t(TokenType::LTE, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    else {
      // < operator
      // push back the last char in the stream
      inputFile.putback(c1);
      // token
      Token t(TokenType::LT, tLexeme);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
  }
  case '>':
```

```
{
  // lexeme
  std::string tLexeme;
  tLexeme.push_back(c);
  inputFile >> c1;
  if (c1 == '=') {
    tLexeme.push_back(c1);
    // token
    Token t(TokenType::GTE, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  else {
    // > operator
    // push back the last char in the stream
    inputFile.putback(c1);
    // token
    Token t(TokenType::GT, tLexeme);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
}
case '.': // can be a constant .y
{
  // lexeme
  std::string tLexeme;
  tLexeme.push_back(c);
  // numerical value
  double xVal = 0.0;
  // fractional part
  double fPart = 0.0;
  double y = 1.0 / 10.0;
  while ((c1 = inputFile.get()) && isdigit(c1)) {
    fPart = fPart + (c1 - '0') * y;
    y /= 10.0;
    tLexeme.push_back(c1);
  }
  // the real value
  xVal = xVal + fPart;
  // push back the last char in the stream
  inputFile.putback(c1);
```

```
    // the token
    Token t(TokenType::REALCONST, tLexeme, xVal);
    // save the current position
    currPosition = inputFile.tellg();
    return t;
  }
  case '0': // 0 constant or 0.x constant
  {
    std::string tLexeme;
    tLexeme.push_back(c);
    // numerical values are stored in these variables
    int iVal = 0;
    double xVal = 0.0;
    inputFile >> c1;
    if (isdigit(c1)) {
      throw Exception("A decimal constant must start with 1-9");
    }
    else if (c1 == '.') { // A real number 0.x
      // the fractional part
      double fPart = 0.0;
      double y = 1.0 / 10.0;
      tLexeme.push_back(c1);
      while ((c1 = inputFile.get()) && isdigit(c1)) {
        fPart = fPart + (c1 - '0') * y;
        y /= 10.0;
        tLexeme.push_back(c1);
      }
      // the real value
      xVal = xVal + fPart;
      // push back the last char in the stream
      inputFile.putback(c1);
      // the token
      Token t(TokenType::REALCONST, tLexeme, xVal);
      // save the current position
      currPosition = inputFile.tellg();
      return t;
    }
    else {  // 0 constant
      // push back the last char in the stream
      inputFile.putback(c1);
      // the token
      Token t(TokenType::INTCONST, tLexeme, iVal);
      // save the current position
```

```
      currPosition = inputFile.tellg();
      return t;
    }
  }
  default:
  {
    if (isdigit(c)) { // a n constant or a x.y constant
      // lexeme
      std::string tLexeme;
      tLexeme.push_back(c);
      // numerical values are stored in these variables
      int iVal = 0;
      double xVal = 0.0;
      // update the values
      iVal = 10 * iVal + c - '0';
      xVal = 10 * xVal + c - '0';
      while ((c1 = inputFile.get()) && isdigit(c1)) { // decimal part
        iVal = 10 * iVal + c1 - '0';
        xVal = 10 * xVal + c1 - '0';
        tLexeme.push_back(c1);
      }
      if (c1 != '.') {  // An integer constant
        // push back the last char in the stream
        inputFile.putback(c1);
        // the token
        Token t(TokenType::INTCONST, tLexeme, iVal);
        // save the current position
        currPosition = inputFile.tellg();
        return t;
      }
      else {  // A real costant x.y
       // the fractional part
        double fPart = 0.0;
        double y = 1.0 / 10.0;
        tLexeme.push_back(c1);
        while ((c1 = inputFile.get()) && isdigit(c1)) {
          fPart = fPart + (c1 - '0') * y;
          y /= 10.0;
          tLexeme.push_back(c1);
        }
        // the real value
        xVal = xVal + fPart;
        // push back the last char in the stream
```

```
        inputFile.putback(c1);
        // the token
        Token t(TokenType::REALCONST, tLexeme, xVal);
        // save the current position
        currPosition = inputFile.tellg();
        return t;
      }
    }
    else if (isalpha(c)) {  // keyword or identifier
      // lexeme
      std::string tLexeme;
      tLexeme.push_back(c);
      // store lexeme characters
      while ((c1 = inputFile.get()) && isalnum(c1)) {
        tLexeme.push_back(c1);
      }
      // push back the last char in the stream
      inputFile.putback(c1);
      // keyword test
      if (keywordMap.find(tLexeme) != keywordMap.end())
      {
        // keyword detected
        // search the keyword token type
        Token t(keywordMap[tLexeme], tLexeme);
        // save the current position
        currPosition = inputFile.tellg();
        return t;
      }
      else {
        // identifier
        // the token - the last tLexeme is the identifier
        Token t(TokenType::IDENT, tLexeme, tLexeme);
        // save the current position
        currPosition = inputFile.tellg();
        return t;
      }

    }
    else {
      throw Exception("Unknown character");
    }
  }
}
```

```
  }
  // end of the file - the special token END
  std::string s("END");
  Token t(TokenType::END, s);
  return t;
}
```

Source0.cpp

```
#include <iostream>
#include <string>

#include "scanner.h"
#include "token.h"
#include "exceptions.h"

std::ofstream outputFile;

int main() {
  std::string fileName;
  std::string outFileName;
  TokenType type;
  std::cout << "Input␣file␣name:␣";
  std::cin >> fileName;
  std::cout << "Output␣file␣name:␣";
  std::cin >> outFileName;
  outputFile.open(outFileName, std::ofstream::out | std::ofstream::app);

  try {
    Scanner sc(fileName);
    do {
      Token t = sc.nextToken();
      t.print(outputFile);
      type = t.getType();
    } while (type != TokenType::END);
  }
  catch (Exception e) {
    e.print();
    return 1;
  }

  return 0;
}
```

3 The parser

3.1 Eliminating left recursion

The grammar is recursive and it cannot be parsed in this form.

The nonterminals ⟨*decl–list*⟩, ⟨*ident–list*⟩ and ⟨*instr–list*⟩ are left recursive. We add three new nonterminal symbols, ⟨*decl–list1*⟩, ⟨*ident–list1*⟩ and ⟨*instr–list1*⟩, and the following six productions:

$$
\begin{aligned}
\langle \textit{decl–list} \rangle &\rightarrow \langle \textit{decl–list} \rangle\ \langle \textit{decl} \rangle \\
\langle \textit{decl–list} \rangle &\rightarrow \epsilon \\
\langle \textit{ident–list} \rangle &\rightarrow \langle \textit{ident–list} \rangle\ \text{','}\ \textbf{\textit{ident}} \\
\langle \textit{ident–list} \rangle &\rightarrow \textbf{\textit{ident}} \\
\langle \textit{instr–list} \rangle &\rightarrow \langle \textit{instr–list} \rangle\ \langle \textit{instr} \rangle \\
\langle \textit{instr–list} \rangle &\rightarrow \epsilon
\end{aligned}
$$

change as follows:

$$
\begin{aligned}
\langle \textit{decl–list} \rangle &\rightarrow \langle \textit{decl–list1} \rangle \\
\langle \textit{decl–list1} \rangle &\rightarrow \langle \textit{decl} \rangle\ \langle \textit{decl–list1} \rangle \\
\langle \textit{decl–list1} \rangle &\rightarrow \epsilon \\
\langle \textit{instr–list} \rangle &\rightarrow \langle \textit{instr–list1} \rangle \\
\langle \textit{instr–list1} \rangle &\rightarrow \langle \textit{instr} \rangle\ \langle \textit{instr–list1} \rangle \\
\langle \textit{instr–list1} \rangle &\rightarrow \epsilon \\
\langle \textit{ident–list} \rangle &\rightarrow \textbf{\textit{ident}}\ \langle \textit{ident–list1} \rangle \\
\langle \textit{ident–list1} \rangle &\rightarrow \text{','}\ \textbf{\textit{ident}}\ \langle \textit{ident–list1} \rangle \\
\langle \textit{ident–list1} \rangle &\rightarrow \epsilon
\end{aligned}
$$

The nonterminals ⟨*expr*⟩ and ⟨*term*⟩ are also left recursive. We add two new nonterminal symbols, ⟨*expr1*⟩ and ⟨*term1*⟩, and the following four productions:

$$
\begin{aligned}
\langle \textit{expr} \rangle &\rightarrow \langle \textit{expr} \rangle\ \text{'+'}\ \langle \textit{term} \rangle \\
\langle \textit{expr} \rangle &\rightarrow \langle \textit{term} \rangle \\
\langle \textit{term} \rangle &\rightarrow \langle \textit{term} \rangle\ \text{'*'}\ \langle \textit{fact} \rangle \\
\langle \textit{term} \rangle &\rightarrow \langle \textit{fact} \rangle
\end{aligned}
$$

change as follows:

$$
\begin{aligned}
\langle expr \rangle &\rightarrow \langle term \rangle \ \langle expr1 \rangle \\
\langle expr1 \rangle &\rightarrow \ '+' \ \langle term \rangle \ \langle expr1 \rangle \\
\langle expr1 \rangle &\rightarrow \epsilon \\
\langle term \rangle &\rightarrow \langle fact \rangle \ \langle term1 \rangle \\
\langle term1 \rangle &\rightarrow \ '*' \ \langle fact \rangle \ \langle term1 \rangle \\
\langle term1 \rangle &\rightarrow \epsilon
\end{aligned}
$$

The new grammar generating the same language, which can be parsed

top-down:

$$
\begin{aligned}
\langle program \rangle &\rightarrow \textbf{int main } '(' ')' \langle block \rangle \\
\langle block \rangle &\rightarrow '\{' \langle content \rangle '\}' \\
\langle content \rangle &\rightarrow \langle decl\text{–}list \rangle \langle instr\text{–}list \rangle \\
\langle content \rangle &\rightarrow \langle instr\text{–}list \rangle \\
\langle decl\text{–}list1 \rangle &\rightarrow \langle decl \rangle \langle decl\text{–}list1 \rangle \\
\langle decl\text{–}list1 \rangle &\rightarrow \epsilon \\
\langle decl \rangle &\rightarrow \langle decl\text{–}type \rangle \langle ident\text{–}list \rangle ';' \\
\langle decl\text{–}type \rangle &\rightarrow \textbf{int} \mid \textbf{double} \\
\langle ident\text{–}list \rangle &\rightarrow \boldsymbol{ident} \langle ident\text{–}list1 \rangle \\
\langle ident\text{–}list1 \rangle &\rightarrow ',' \boldsymbol{ident} \langle ident\text{–}list1 \rangle \\
\langle ident\text{–}list1 \rangle &\rightarrow \epsilon \\
\langle instr\text{–}list \rangle &\rightarrow \langle instr\text{–}list1 \rangle \\
\langle instr\text{–}list1 \rangle &\rightarrow \langle instr \rangle \langle instr\text{–}list1 \rangle \\
\langle instr\text{–}list1 \rangle &\rightarrow \epsilon \\
\langle instr \rangle &\rightarrow \langle instr\text{–}assign \rangle \mid \langle instr\text{–}return \rangle \mid \langle instr\text{–}if \rangle \mid \langle instr\text{–}empty \rangle \\
\langle instr\text{–}assign \rangle &\rightarrow \boldsymbol{ident} '=' \langle expr \rangle ';' \\
\langle instr\text{–}return \rangle &\rightarrow \textbf{return } \langle expr \rangle ';' \\
\langle instr\text{–}empty \rangle &\rightarrow ';' \\
\langle expr \rangle &\rightarrow \langle term \rangle \langle expr1 \rangle \\
\langle expr1 \rangle &\rightarrow '+' \langle term \rangle \langle expr1 \rangle \\
\langle expr1 \rangle &\rightarrow \epsilon \\
\langle term \rangle &\rightarrow \langle fact \rangle \langle term1 \rangle \\
\langle term1 \rangle &\rightarrow '*' \langle fact \rangle \langle term1 \rangle \\
\langle term1 \rangle &\rightarrow \epsilon \\
\langle fact \rangle &\rightarrow '(' \langle expr \rangle ')' \\
\langle fact \rangle &\rightarrow \boldsymbol{ident} \\
\langle fact \rangle &\rightarrow \boldsymbol{int\text{–}const} \\
\langle fact \rangle &\rightarrow \boldsymbol{real\text{–}const} \\
\langle instr\text{–}if \rangle &\rightarrow \textbf{if } '(' \langle cond \rangle ')' \langle instr\text{–}block \rangle \textbf{ else } \langle instr\text{–}block \rangle \\
\langle instr\text{–}block \rangle &\rightarrow \langle instr \rangle \mid \langle block \rangle \\
\langle cond \rangle &\rightarrow \langle rel\text{–}operand \rangle \langle rel\text{–}operator \rangle \langle rel\text{–}operand \rangle \\
\langle rel\text{–}operand \rangle &\rightarrow \boldsymbol{ident} \mid \boldsymbol{int\text{–}const} \mid \boldsymbol{real\text{–}const} \\
\langle rel\text{–}operator \rangle &\rightarrow '<' \mid '>' \mid '==' \mid '>=' \mid '<=' \mid '! ='
\end{aligned}
$$

For each nonterminal symbol, a function (with the same name) has to be written.

The new set of nonterminal symbols:

- $\langle program \rangle$
- $\langle block \rangle$
- $\langle content \rangle$
- $\langle decl\text{–}list \rangle$
- $\langle decl\text{–}list1 \rangle$
- $\langle decl \rangle$
- $\langle decl\text{–}type \rangle$
- $\langle ident\text{–}list \rangle$
- $\langle ident\text{–}list1 \rangle$
- $\langle instr\text{–}block \rangle$
- $\langle instr \rangle$
- $\langle instr\text{–}list \rangle$
- $\langle instr\text{–}list1 \rangle$
- $\langle instr\text{–}return \rangle$
- $\langle instr\text{–}assign \rangle$
- $\langle instr\text{–}empty \rangle$
- $\langle instr\text{–}if \rangle$
- $\langle expr \rangle$
- $\langle expr1 \rangle$
- $\langle term \rangle$
- $\langle term1 \rangle$
- $\langle fact \rangle$

- $\langle cond \rangle$
- $\langle rel\text{–}operand \rangle$
- $\langle rel\text{–}operator \rangle$

3.2 The *First* sets for non-terminal symbols

Because in the recursive descent parsing the *lookahead* symbol is not used, it will be replaced with the terminal symbols returned by the $First()$ function.

- $\textbf{First}(\langle program \rangle) = \{\textbf{int}\}$
- $\textbf{First}(\langle block \rangle) = \{'\{'\}$
- $\textbf{First}(\langle decl\text{–}list \rangle) = \textbf{First}(\langle decl\text{–}list1 \rangle) = \textbf{First}(\langle decl \rangle) \cup \{\epsilon\} = \textbf{First}(\langle decl\text{–}type \rangle) \cup \{\epsilon\} = \{\textbf{int},\ \textbf{double},\ \epsilon\}$
- $\textbf{First}(\langle ident\text{–}list \rangle) = \{\boldsymbol{ident}\}$
- $\textbf{First}(\langle ident\text{–}list1 \rangle) = \{',' \ \epsilon\}$
- $\textbf{First}(\langle instr\text{–}block \rangle) = \{'\{',\ \textbf{return},\ \textbf{if},\ \boldsymbol{ident},\ ';'\}$
- $\textbf{First}(\langle instr\text{–}list \rangle) = \textbf{First}(\langle instr\text{–}list1 \rangle) = \textbf{First}(\langle instr \rangle) \cup \{\epsilon\} = \{\textbf{return},\ \textbf{if},\ \boldsymbol{ident},\ ';',\ \epsilon\}$
- $\textbf{First}(\langle content \rangle) = \textbf{First}(\langle decl\text{–}list \rangle) \cup \textbf{First}(\langle instr\text{–}list1 \rangle) = \{\textbf{int},\ \textbf{double},\ \textbf{return},\ \textbf{if},\ \boldsymbol{ident},\ ';',\ \epsilon\}$
- $\textbf{First}(\langle instr\text{–}return \rangle) = \{\textbf{return}\}$
- $\textbf{First}(\langle instr\text{–}assign \rangle) = \{\boldsymbol{ident}\}$
- $\textbf{First}(\langle instr\text{–}empty \rangle) = \{';'\}$
- $\textbf{First}(\langle instr\text{–}if \rangle) = \{\textbf{if}\}$
- $\textbf{First}(\langle expr \rangle) = \textbf{First}(\langle term \rangle) = \textbf{First}(\langle fact \rangle) = \{'(',\ \boldsymbol{ident}, \boldsymbol{int\text{–}const},\ \boldsymbol{real\text{–}const}\}$
- $\textbf{First}(\langle expr1 \rangle) = \{'+'\}$
- $\textbf{First}(\langle term1 \rangle) = \{'*'\}$
- $\textbf{First}(\langle cond \rangle) = \textbf{First}(\langle rel\text{–}operand \rangle) = \{\boldsymbol{ident}, \boldsymbol{int\text{–}const},\ \boldsymbol{real\text{–}const}\}$
- $\textbf{First}(\langle rel\text{–}operator \rangle) = \{'<',\ '>',\ ,\ '==',\ '<=',\ '>=',\ '!='\}$

3.3 Source files

parser.h

```
#pragma once

#ifndef PARS
#define PARS

#include <string>
#include <memory>
#include "scanner.h"
#include "toktype.h"

class Parser {
  std::string fileName;
  std::unique_ptr<Scanner> scanner;
  TokenType tokType;
public:
  Parser(std::string name);
  void parse();
private:
  void match(TokenType tt);
  void program();
  void decl_list();
  void decl_list1();
  void decl();
  void content();
  void decl_type();
  void ident_list();
  void ident_list1();
  void instr_block();
  void block();
  void instr_list();
  void instr_list1();
  void instr();
  void instr_return();
  void instr_assign();
  void instr_empty();
  void instr_if();
  void expr();
  void term();
  void fact();
  void expr1();
```

```
  void term1();
  void cond();
  void rel_operand();
  void rel_operator();
};

#endif
```

parser.cpp

```
#include "parser.h"
#include "exceptions.h"

#include <string>
#include <iostream>

Parser::Parser(std::string name) :
  scanner(std::make_unique<Scanner>(name)) {
  tokType = scanner->nextToken().getType();
}

void Parser::match(TokenType tt) {
  if (tokType == tt) {
    Token tok = scanner->nextToken();
    tokType = tok.getType();
  }
  else if (tokType == TokenType::END) {
    std::cout << "End of file !!\n";
    exit(1);
  } else {
    throw Exception("Illegal token type !");
  }
}

void Parser::parse() {
  program();
  std::cout << "The program is syntactically correct\n";
}

void Parser::program() {
  switch (tokType) {
  case TokenType::INT:
    match(TokenType::INT);
    match(TokenType::MAIN);
```

```
    match(TokenType::LPAREN);
    match(TokenType::RPAREN);
    block();
    break;
  default:
    throw Exception("'program': Illegal token!");
  }
}

void Parser::decl_list() {
  switch (tokType) {
  case TokenType::INT:
  case TokenType::DOUBLE:
    decl_list1();
    break;
  case TokenType::END:
    break;
  default:
    throw Exception("'decl_list': Illegal token!");
  }
}

void Parser::decl_list1() {
  switch (tokType) {
  case TokenType::INT:
  case TokenType::DOUBLE:
    decl();
    decl_list1();
    break;
  default:
    break;  // not throwing any exception - case <decl-list1> -> epsilon
  }
}

void Parser::decl() {
  switch (tokType) {
  case TokenType::INT:
  case TokenType::DOUBLE:
    decl_type();
    ident_list();
    match(TokenType::SEMICOLON);
    break;
  default:
```

```
    throw Exception("'decl': Illegal token !");
  }
}
void Parser::decl_type() {
  switch (tokType) {
  case TokenType::INT:
    match(TokenType::INT);
    break;
  case TokenType::DOUBLE:
    match(TokenType::DOUBLE);
    break;
  default:
    throw Exception("'decl_type': Illegal token !");
  }
}

void Parser::ident_list() {
  switch (tokType) {
  case TokenType::IDENT:
    match(TokenType::IDENT);
    ident_list1();
    break;
  default:
    throw Exception("'ident_list': Illegal token !");
  }
}

void Parser::ident_list1() {
  switch (tokType) {
  case TokenType::COMMA:
    match(TokenType::COMMA);
    match(TokenType::IDENT);
    ident_list1();
    break;
  default:
    break;  // not throwing any exception - case <ident-list1> -> epsilo
  }
}

void Parser::instr_block() {
  switch (tokType) {
  case TokenType::LBRACE:
    block();
```

```
    break;
  case TokenType::RETURN:
  case TokenType::IF:
  case TokenType::IDENT:
  case TokenType::SEMICOLON:
    instr();
    break;
  default:
    throw Exception("'instr_block': Illegal token !");
  }
}

void Parser::block() {
  switch (tokType) {
  case TokenType::LBRACE:
    match(TokenType::LBRACE);
    content();
    match(TokenType::RBRACE);
    break;
  default:
    throw Exception("'block': Illegal token !");
  }
}

void Parser::content() {
  switch (tokType) {
  case TokenType::INT:
  case TokenType::DOUBLE:
    decl_list();
    instr_list();
    break;
  case TokenType::RETURN:
  case TokenType::IF:
  case TokenType::IDENT:
  case TokenType::SEMICOLON:
    instr_list();
    break;
  default:
    throw Exception("'content': Illegal token !");
  }
}
```

```
void Parser::instr_list() {
  switch (tokType) {
  case TokenType::RETURN:
  case TokenType::IF:
  case TokenType::IDENT:
  case TokenType::SEMICOLON:
    instr_list1();
    break;
  case TokenType::END:
    break;
  default:
    throw Exception("'instr_list': Illegal token!");
  }
}

void Parser::instr_list1() {
  switch (tokType) {
  case TokenType::RETURN:
  case TokenType::IF:
  case TokenType::IDENT:
  case TokenType::SEMICOLON:
    instr();
    instr_list1();
    break;
  default:
    break;  // not throwing any exception - case <instr-list1> -> epsilo
  }
}

void Parser::instr() {
  switch (tokType) {
  case TokenType::RETURN:
    instr_return();
    break;
  case TokenType::IF:
    instr_if();
    break;
  case TokenType::IDENT:
    instr_assign();
    break;
  case TokenType::SEMICOLON:
    instr_empty();
    break;
```

```
  default:
    throw Exception("'instr': Illegal token !");
  }
}

void Parser::instr_return() {
  switch (tokType) {
  case TokenType::RETURN:
    match(TokenType::RETURN);
    expr();
    match(TokenType::SEMICOLON);
    break;
  default:
    throw Exception("'instr_return': Illegal token !");
  }
}

void Parser::instr_assign() {
  switch (tokType) {
  case TokenType::IDENT:
    match(TokenType::IDENT);
    match(TokenType::ASSIGN);
    expr();
    match(TokenType::SEMICOLON);
    break;
  default:
    throw Exception("'instr_assign': Illegal token !");
  }
}

void Parser::instr_empty() {
  switch (tokType) {
  case TokenType::SEMICOLON:
    match(TokenType::SEMICOLON);
    break;
  default:
    throw Exception("'instr_empty': Illegal token !");
  }
}

void Parser::instr_if() {
  switch (tokType) {
  case TokenType::IF:
```

```
    match(TokenType::IF);
    match(TokenType::LPAREN);
    cond();
    match(TokenType::RPAREN);
    instr_block();
    match(TokenType::ELSE);
    instr_block();
    break;
  default:
    throw Exception("'instr_if': Illegal token !");
  }
}

void Parser::expr() {
  switch (tokType) {
  case TokenType::LPAREN:
  case TokenType::IDENT:
  case TokenType::INTCONST:
  case TokenType::REALCONST:
    term();
    expr1();
    break;
  default:
    throw Exception("'expr': Illegal token !");
  }
}

void Parser::term() {
  switch (tokType) {
  case TokenType::LPAREN:
  case TokenType::IDENT:
  case TokenType::INTCONST:
  case TokenType::REALCONST:
    fact();
    term1();
    break;
  default:
    throw Exception("'term': Illegal token !");
  }
}

void Parser::fact() {
  switch (tokType) {
```

```
  case TokenType::LPAREN:
    match(TokenType::LPAREN);
    expr();
    match(TokenType::RPAREN);
    break;
  case TokenType::IDENT:
    match(TokenType::IDENT);
    break;
  case TokenType::INTCONST:
    match(TokenType::INTCONST);
    break;
  case TokenType::REALCONST:
    match(TokenType::REALCONST);
    break;
  default:
    throw Exception("'fact': Illegal token !");
  }
}

void Parser::expr1() {
  switch (tokType) {
  case TokenType::PLUS:
    match(TokenType::PLUS);
    term();
    expr1();
    break;
  default:
    break;  // not throwing any exception - case <expr1> -> epsilon
  }
}

void Parser::term1() {
  switch (tokType) {
  case TokenType::TIMES:
    match(TokenType::TIMES);
    fact();
    term1();
    break;
  default:
    break;  // not throwing any exception - case <term1> -> epsilon
  }
}
```

```
void Parser::cond() {
  switch (tokType) {
  case TokenType::IDENT:
  case TokenType::INTCONST:
  case TokenType::REALCONST:
    rel_operand();
    rel_operator();
    rel_operand();
    break;
  default:
    throw Exception("'cond': Illegal token!");
  }
}

void Parser::rel_operand() {
  switch (tokType) {
  case TokenType::IDENT:
    match(TokenType::IDENT);
    break;
  case TokenType::INTCONST:
    match(TokenType::INTCONST);
    break;
  case TokenType::REALCONST:
    match(TokenType::REALCONST);
    break;
  default:
    throw Exception("'rel_operand': Illegal token!");
  }
}

void Parser::rel_operator() {
  switch (tokType) {
  case TokenType::LT:
    match(TokenType::LT);
    break;
  case TokenType::LTE:
    match(TokenType::LTE);
    break;
  case TokenType::GT:
    match(TokenType::GT);
    break;
  case TokenType::GTE:
    match(TokenType::GTE);
```

```
    break;
  case TokenType::EQ:
    match(TokenType::EQ);
    break;
  case TokenType::NEQ:
    match(TokenType::NEQ);
    break;
  default:
    throw Exception("'rel_operator': Illegal token !");
  }
}
```

Source1.cpp

```
#include <iostream>
#include <string>

#include "scanner.h"
#include "token.h"
#include "parser.h"
#include "exceptions.h"

int main() {
  std::string fileName;
  std::cout << "Input file name: ";
  std::cin >> fileName;

  try {
    Parser parser(fileName);
    parser.parse();
  }
  catch (Exception e) {
    e.print();
    return 1;
  }

  return 0;
}
```