**编译原理课设报告**

－ 简单编译器的实现

软件1404 王若璇 u201417245

**目录：**

1.总体设计思想

2.详细算法设计

3.流程框图

4.运行环境以及相关说明

5.运行的输入输出

6.程序运行结果

7.编译器的使用说明

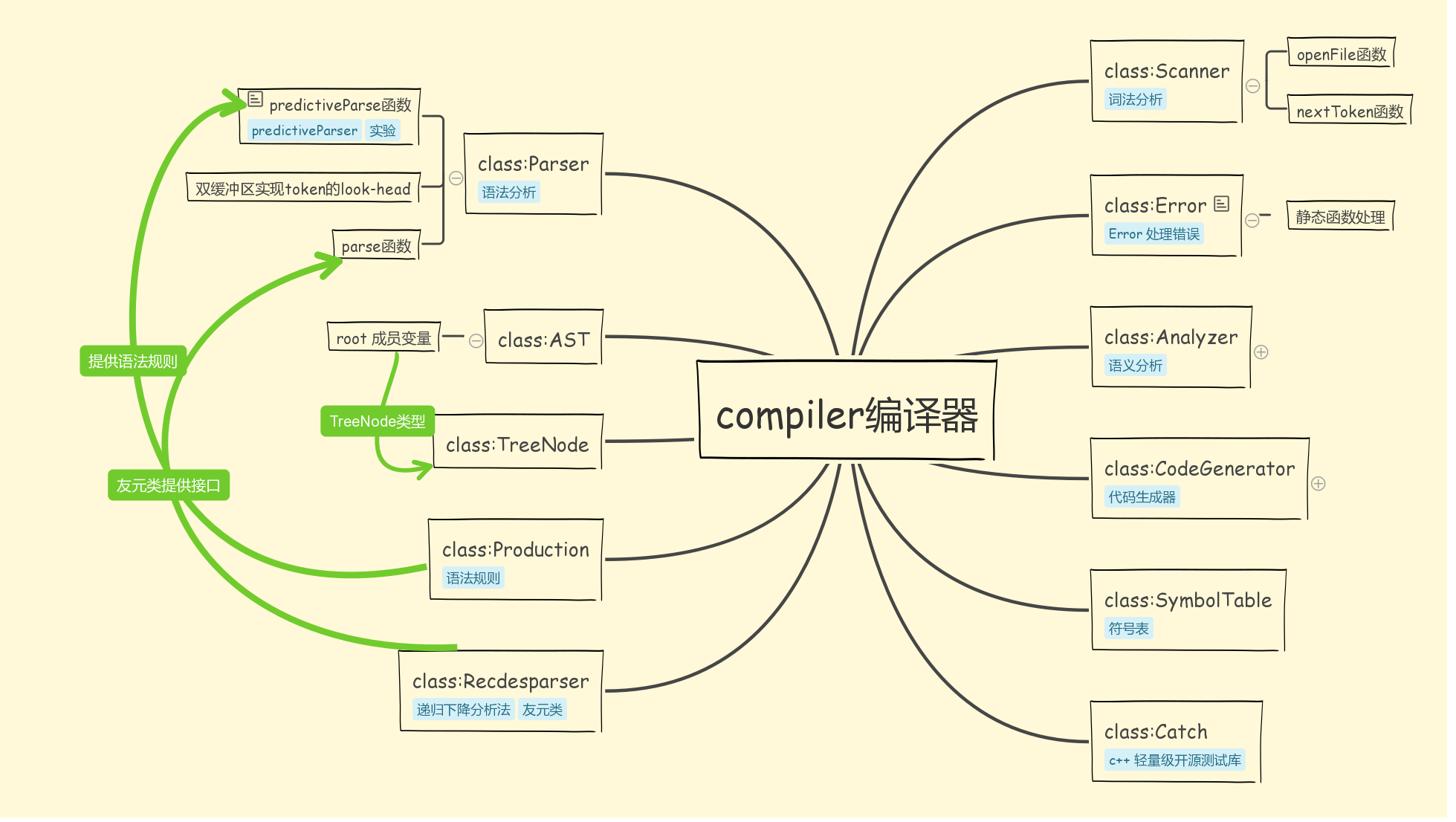
8.亮点

9.心得体会

10.附录代码

**1.总体设计思想**

总体设计思想以及软件架构图如下。（其中CodeGenerator由interCode代替）



compiler 编译器由多个类构成，软件设计采用了面向对象的开发思想，有利于软件的拓展和维护。

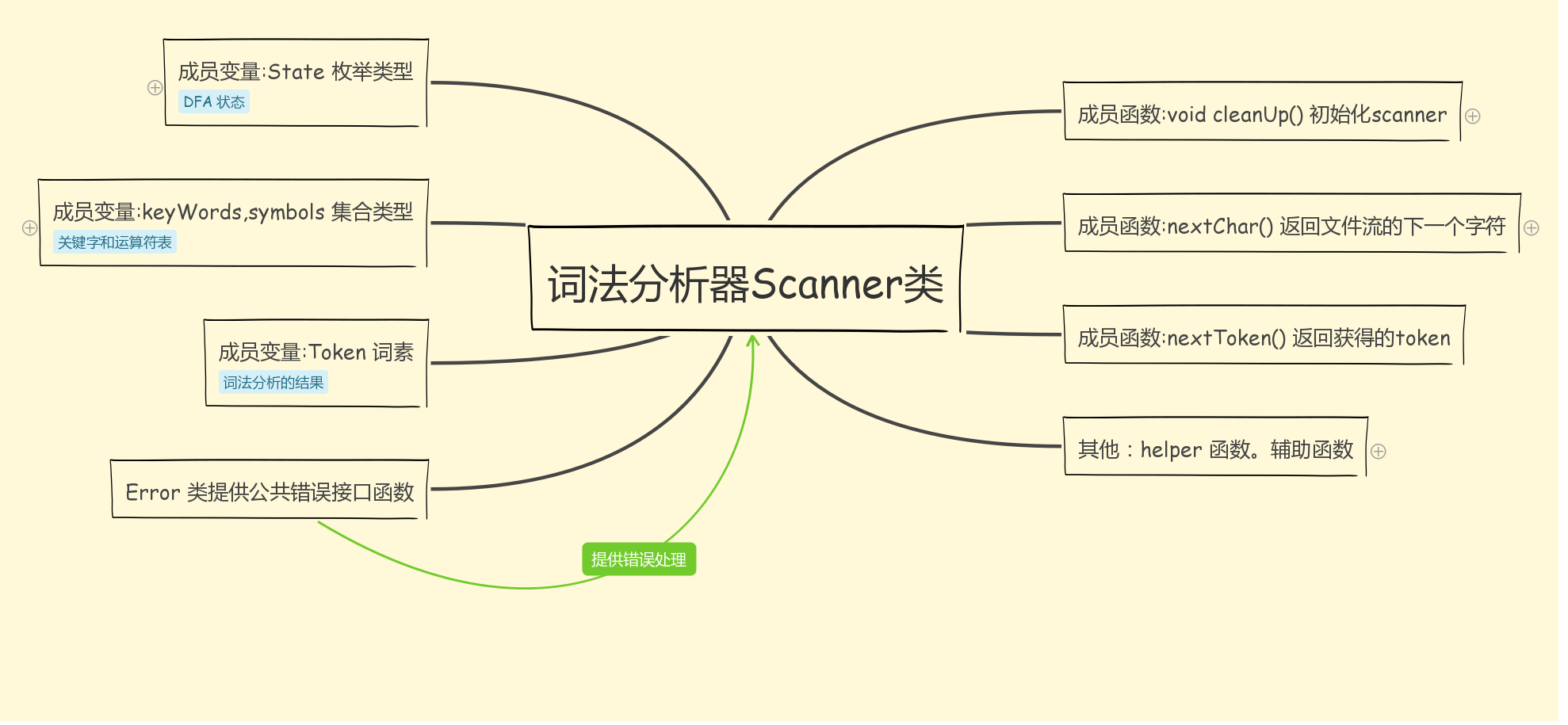
基本设计思想如下：

1. 由词法分析器对源文件进行词法分析，得到若干个词素（tokens）。
2. 由分析出来的词素，进行语法分析，得到一颗抽象的语法树。
3. 在语法树上面进行语义分析，得到中间代码。

注：在上述步骤中同时进行了错误处理。

**2.详细算法设计**

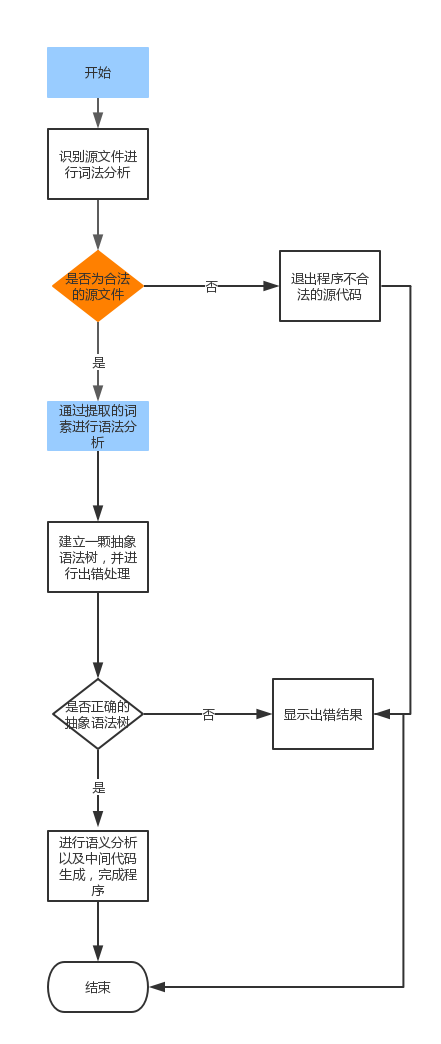
1. 采用了dfa识别源文件。
2. 通过定义不同叶节点的类型值，由递归下降的语法分析方法构造了一棵抽象的语法树。
3. 再由语法制导的语义分析得到中间代码。



词法分析器结构

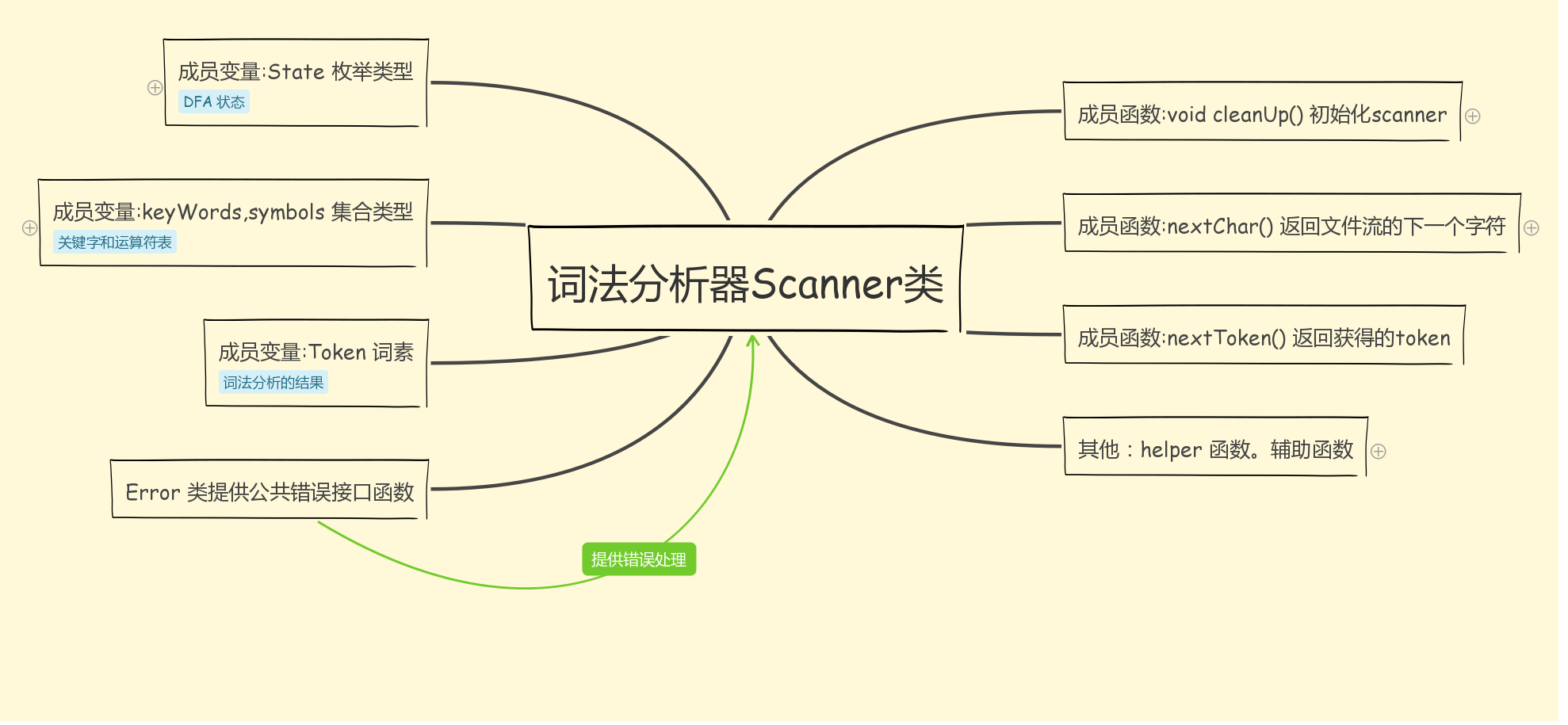
详细细节参照代码，有十分详细的注释。

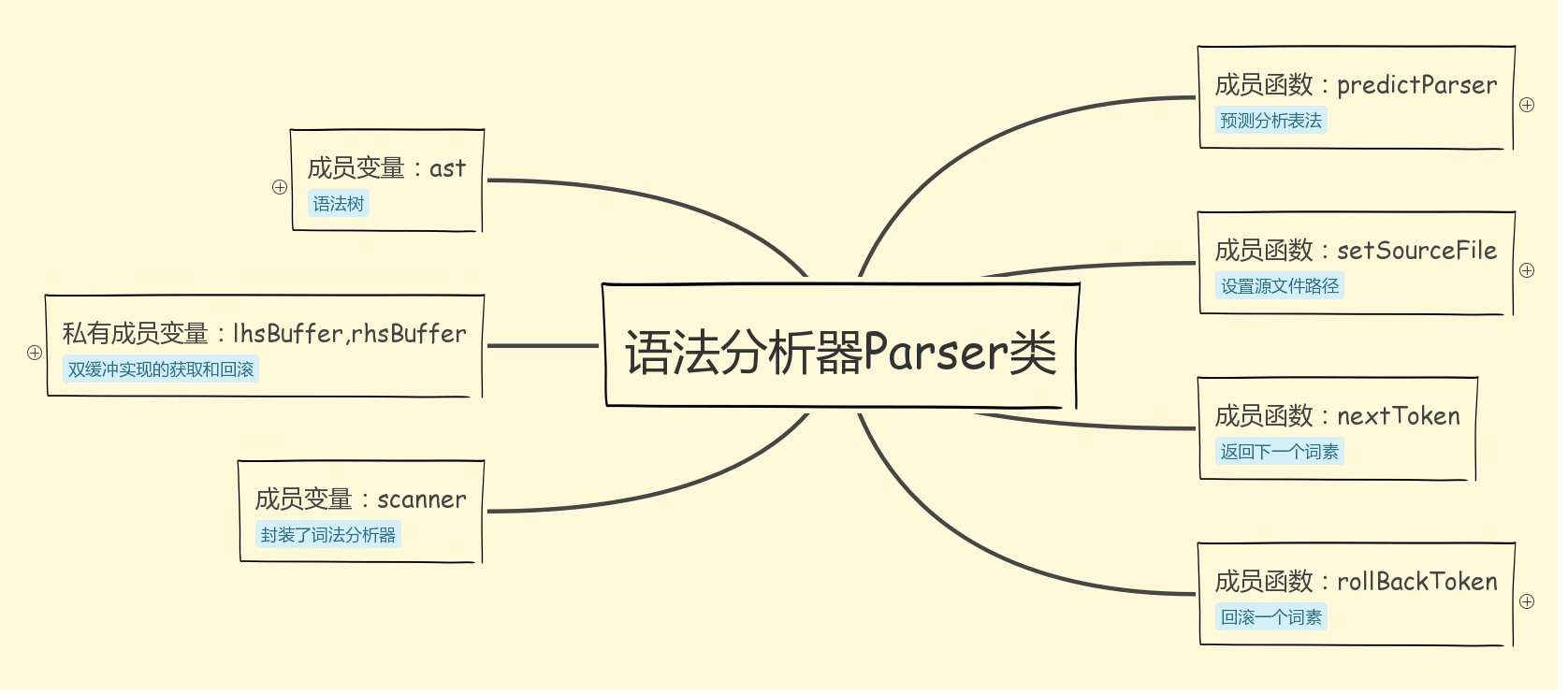
**3.流程框图**



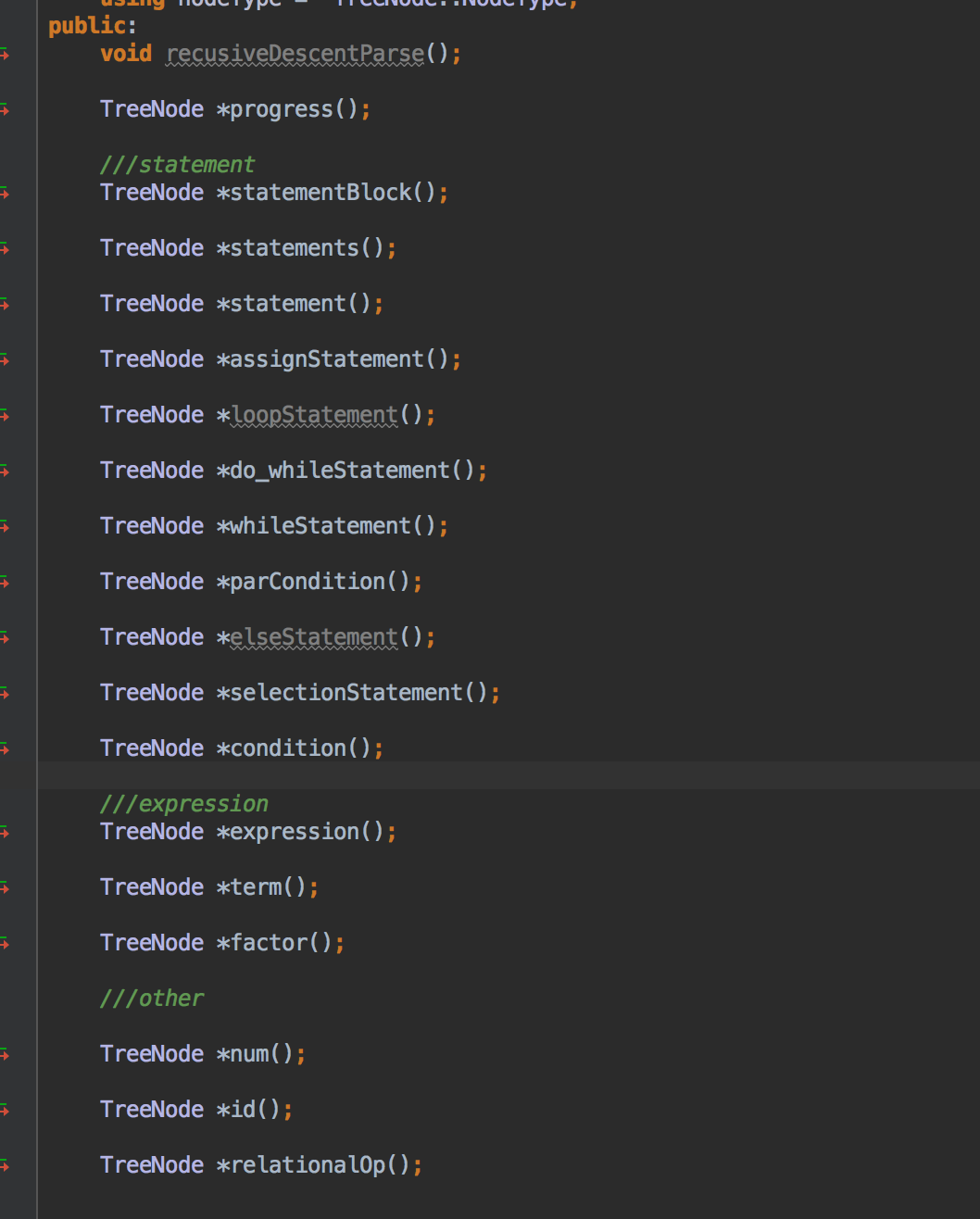
(1)总体流程图。

**4.相关函数说明**

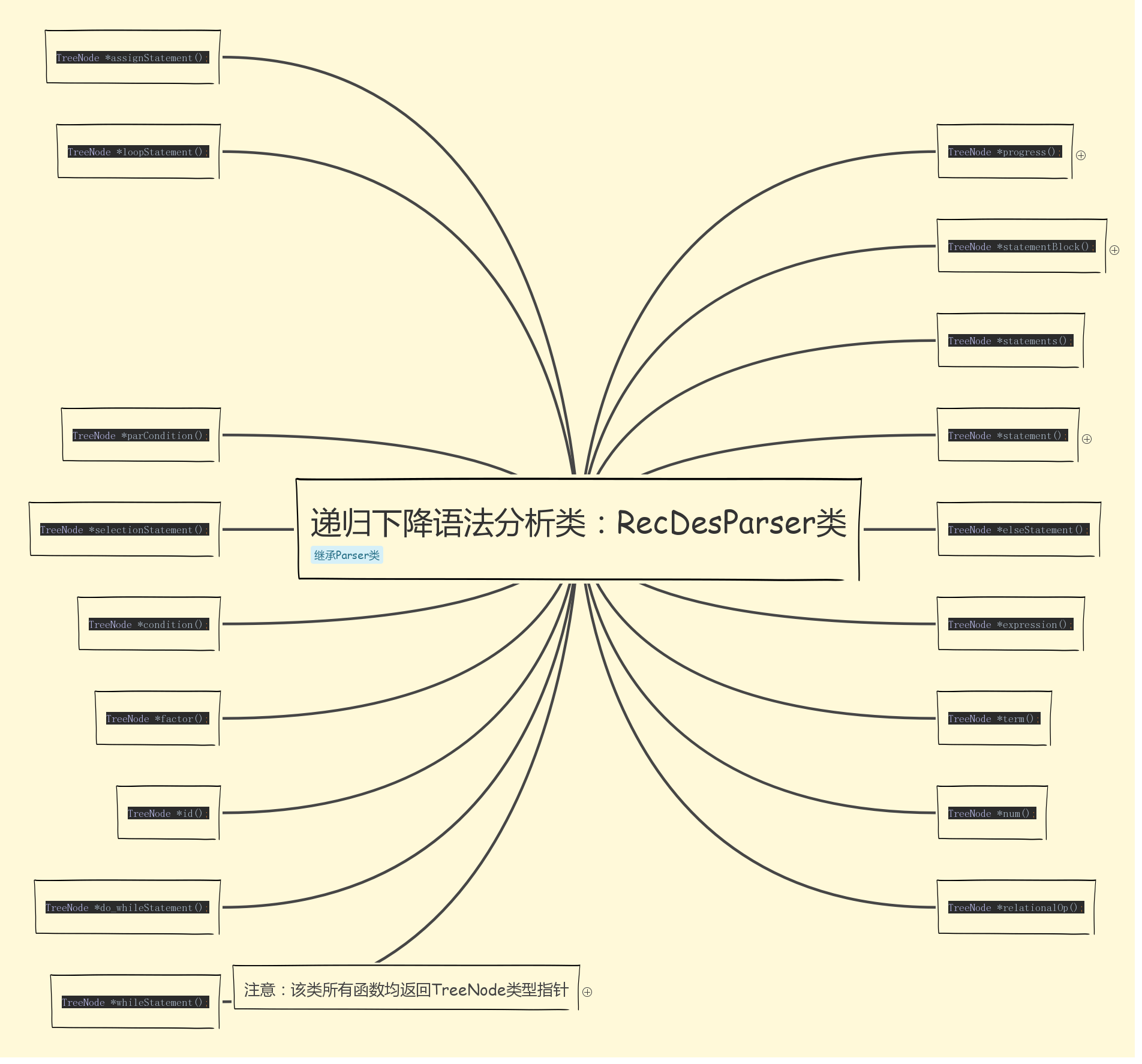
(1)词法分析器说明

(2)语法分析器基类说明。

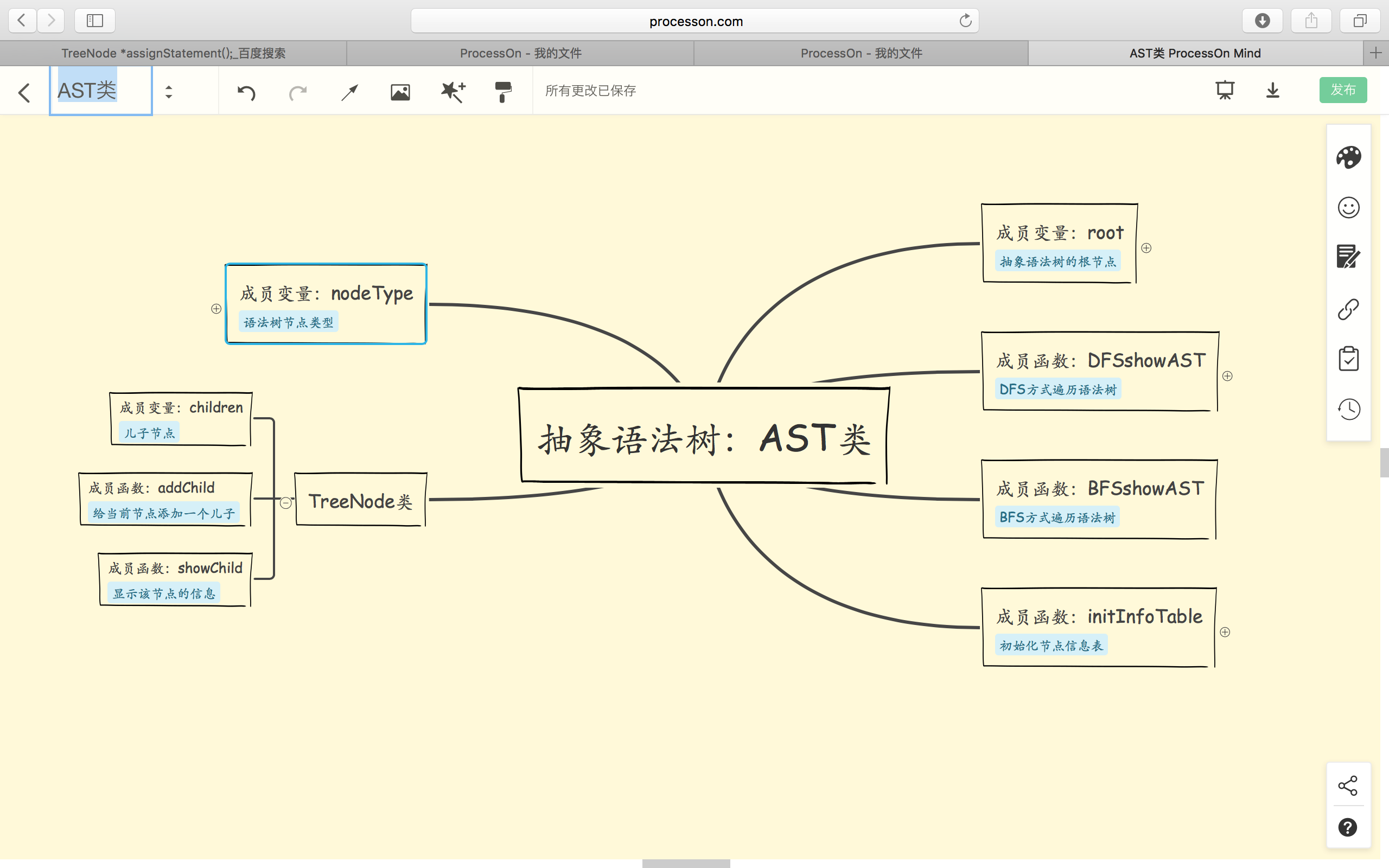
(3)递归下降的语法分析器说明。

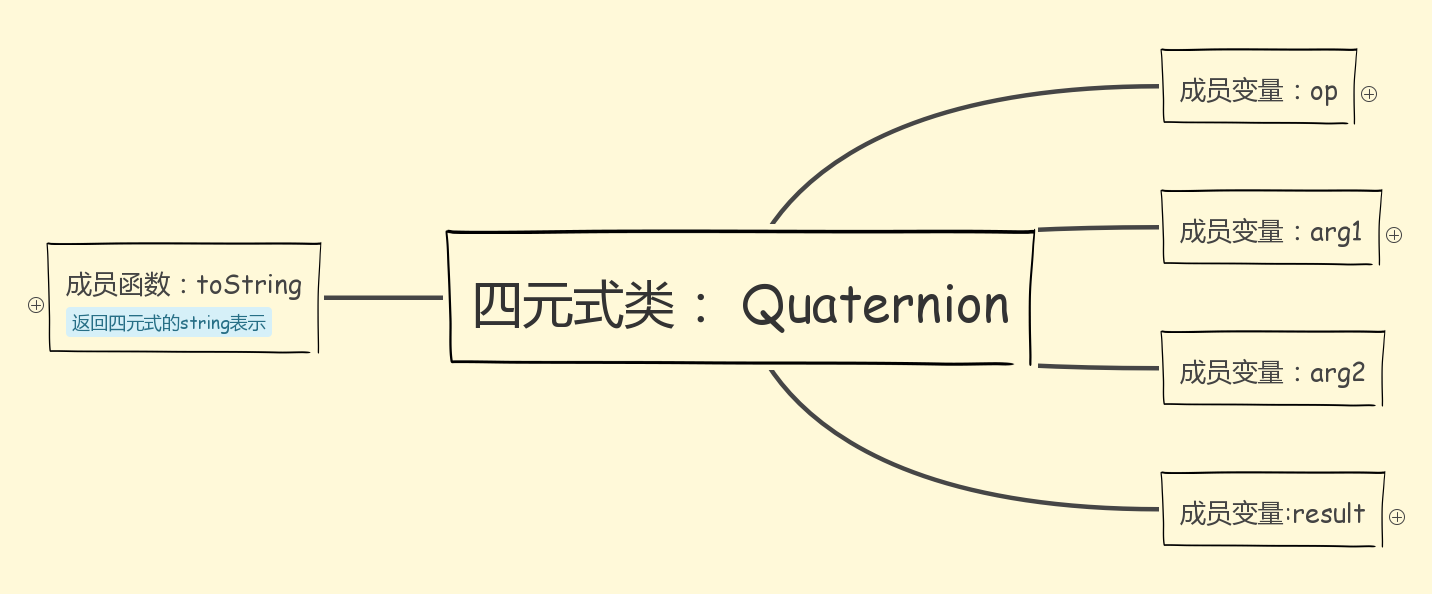
主要函数说明：

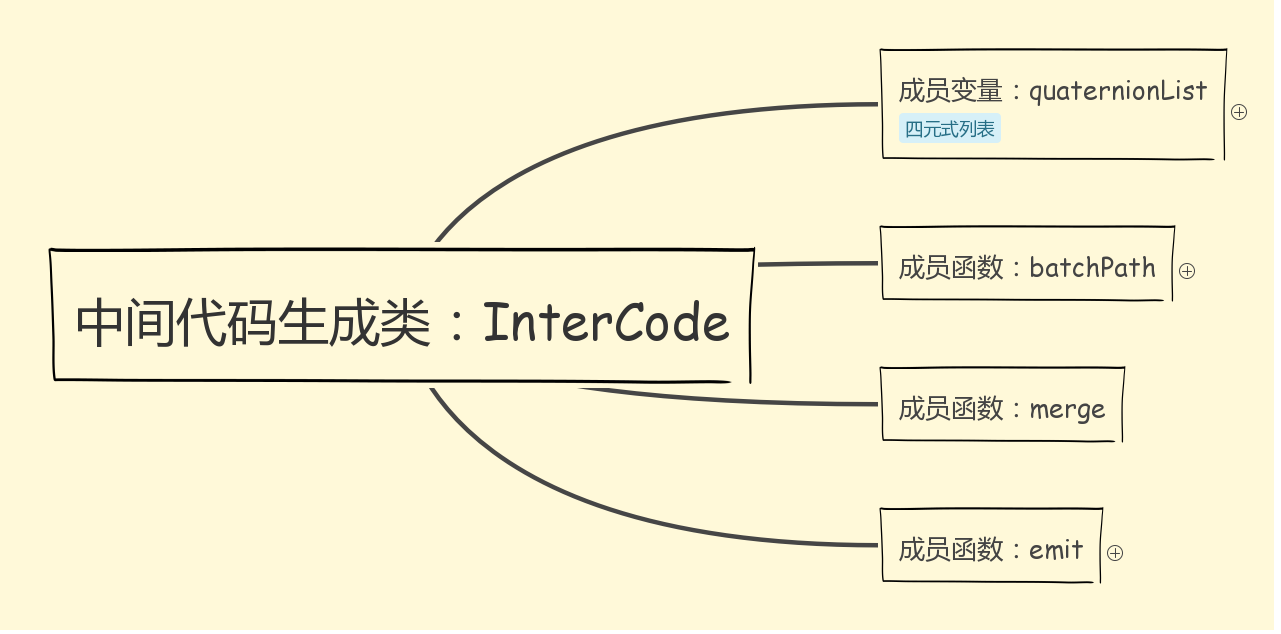
上图是实际实现抽象出来的语法树的节点。

结构说明：

(4)抽象语法树类



(5)中间代码生成类和四元式类



InterCode类中其他的函数：

（6）错误处理类Error

简单的错误输出函数。

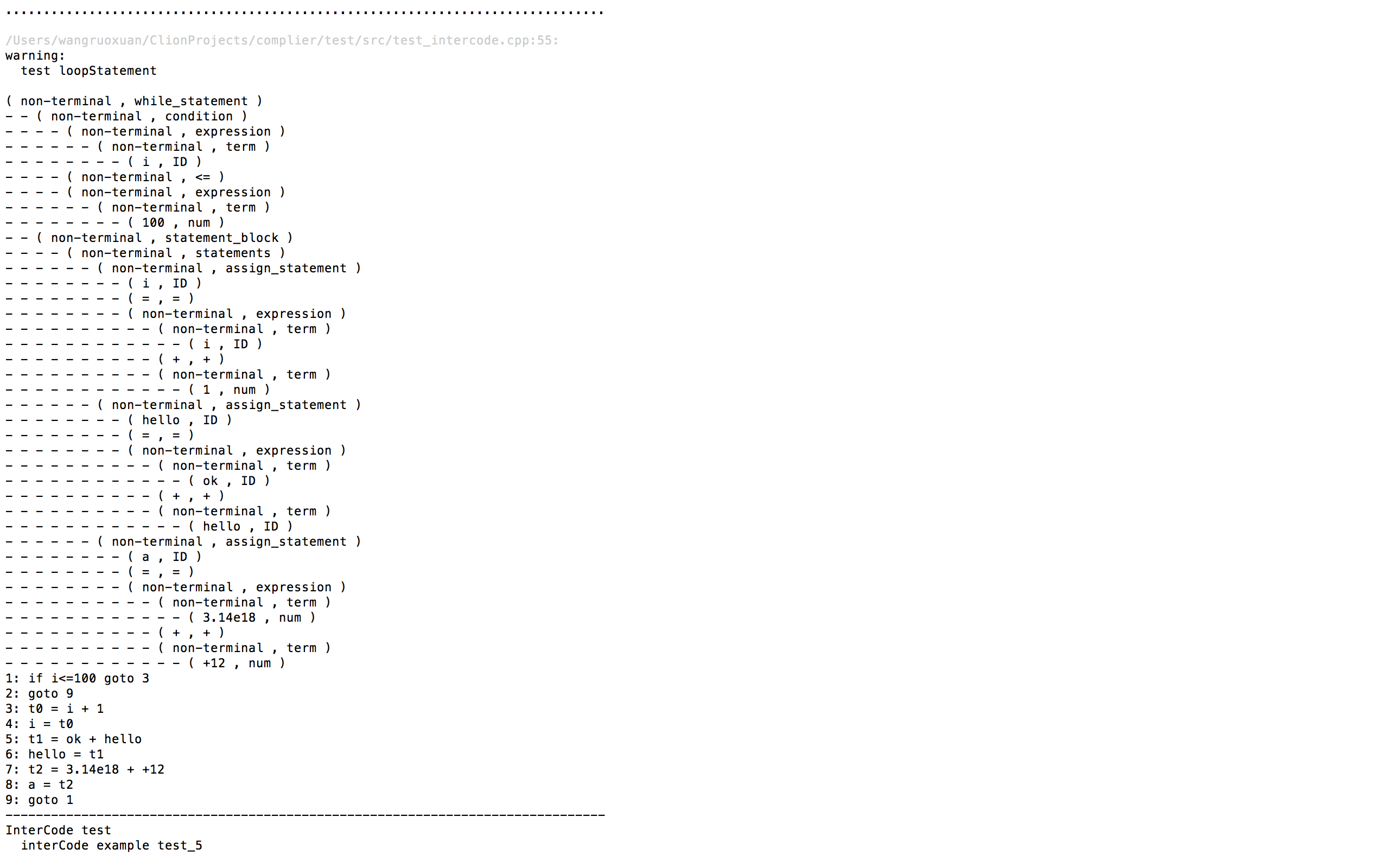
略。

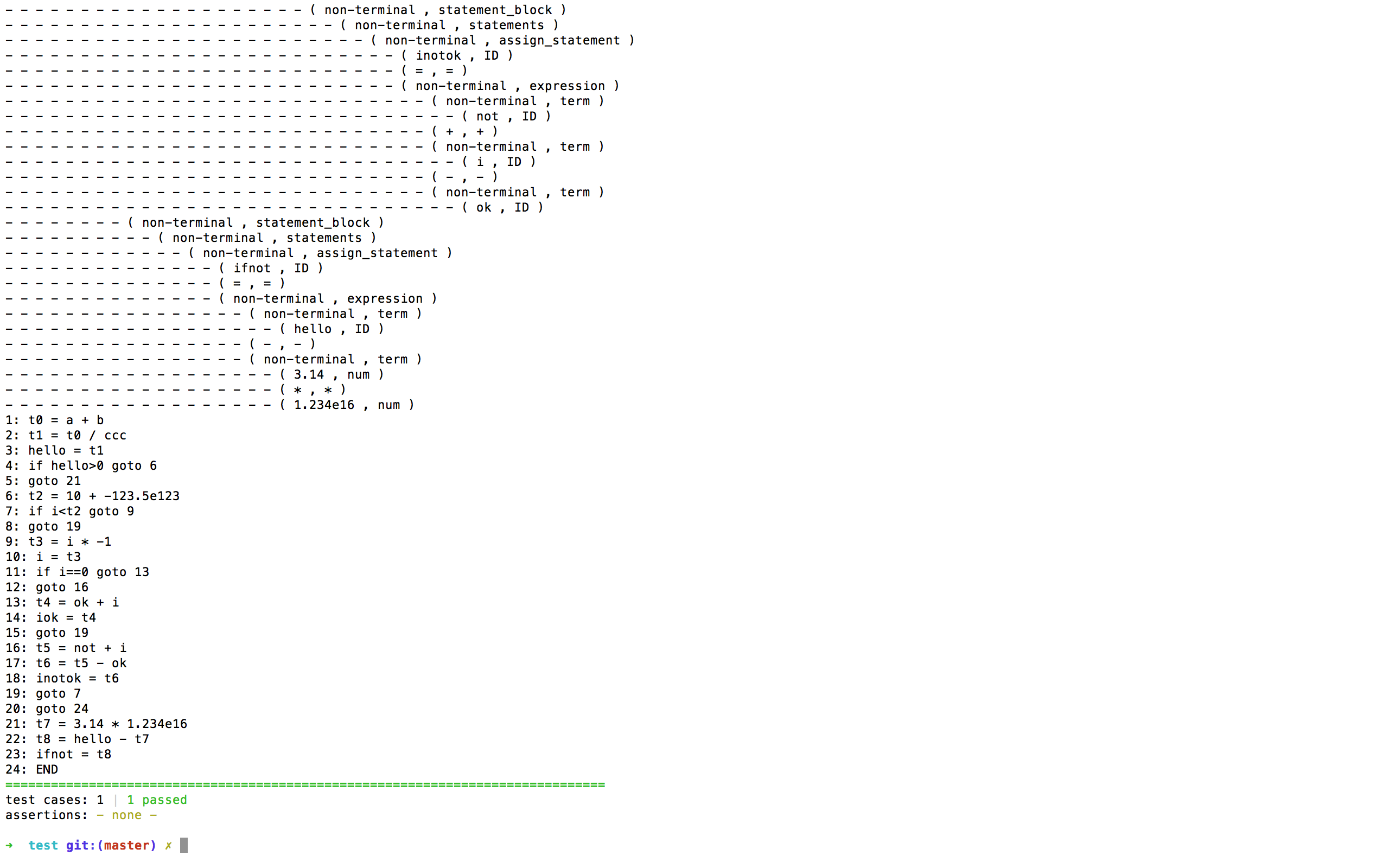
**5.运行的输入输出**

输入：文件输入，位于工程的test／data目录下

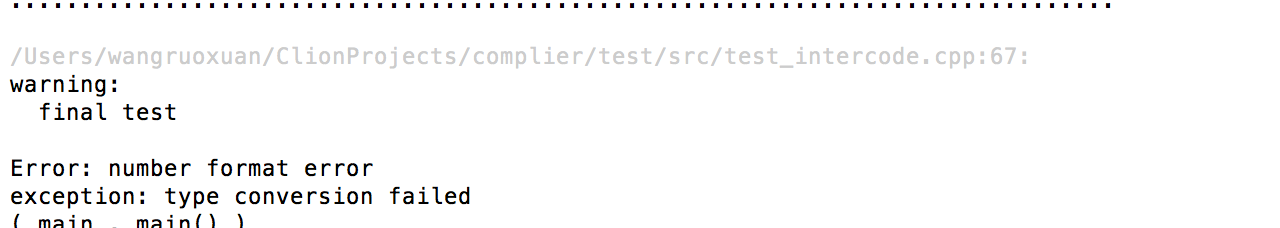
输出：控制台显示，可以设置为文件输出

**6.运行结果截图（部分）**

**正确结果：**



错误结果：

语法错误。

出错会有相应的错误提示，测试样例位于工程项目下，test的data里。可以对照test的src。单元测试源文件进行不同的测试和修改。

**7.编译器的使用说明**

使用cmake编译源代码即可，或者使用clion2016，导入工程源文件，编译。

测试使用了catch单元测试框架。在test目录下。

**8.亮点**

(1)完成了所有功能，功能上面无bug。

(2)使用面向对象的开发思想，利于维护和扩展

(3)代码结构清晰，有十分完备的注释，方便阅读和维护。

(4)测试使用catch单元测试，完整的测试方案，可以测试不同的情况。

**9.心得和体会**

通过本次课设，我不仅学会了编译器的相关知识，而且还学会了思考，如何构建编译器的整体框架。将所学的运用到代码中。提高了代码能力和思维的能力，受益匪浅。

附录代码：

完整工程结构见电子档。

仅列出核心代码。

//

// Created by 王若璇 on 16/12/22.

//

#ifndef COMPILER\_INTERCODE\_H

#define COMPILER\_INTERCODE\_H

#define SHOW\_ROW\_NUM

#define SHOW\_LABEL

#include <iostream>

#include <fstream>

#include <vector>

#include <string>

#include <forward\_list>

#include "AST.h"

using namespace std;

class Quaternion {

public:

string op;

string arg1;

string arg2;

int row;

int chain{-1};

Quaternion(const string &op,

const string &arg1,

const string &arg2,

const string &result) : op(op),

arg1(arg1),

arg2(arg2),

result(result) {

tolNum++;

row = tolNum;

isCondition = false;

index\_ = row - 1;

}

string result;

bool isCondition{false};

bool assigned{false};

bool isJump{false};

int trueList{-1};

int falseList{-1};

int beginCode{-1};

static int tolLabel;

static int tolNum;

private:

int index\_;

public:

int getIndex\_() const;

void setIndex\_(int index\_);

public:

Quaternion() {

tolNum++;

row = tolNum;

isCondition = false;

index\_ = row - 1;

}

///@todo please use share\_ptr

~Quaternion() {/\*cerr<<"tolnum-- "<<tolNum<<endl;tolNum--;\*/};

string toString() const {

if (op == "+" || op == "\*" || op == "/" || op == "-") {

return result + " = " + arg1 + " " + op + " " + arg2;

} else if (op == "=") {

return result + " = " + arg1;

} else if (isCondition) {

///assigned or has't assigned

if (assigned) {

return "if " + arg1 + " goto " + result;

} else {

return "if " + arg1 + " goto 0";

}

} else if (isJump) {

if (assigned) {

return "goto " + result;

} else {

return "goto 0";

}

} else {

return result;

}

}

};

class InterCode {

public:

using nodeType = TreeNode::NodeType;

using E\_place = std::string;

using T\_place = std::string;

using F\_place = std::string;

using codeIndex = int;

static int tolTempNum;

private:

///use a file to save output result.

ofstream fout;

vector<Quaternion> quaternionList\_;

public:

const vector<Quaternion> &getQuaternionList\_() const;

void setQuaternionList\_(const vector<Quaternion> &quaternionList\_);

private:

void bathPatch\_(codeIndex index, int pos);

codeIndex merge\_(codeIndex p1, codeIndex p2);

public:

InterCode();

void emit(const string &str);

void emit(const Quaternion &quaternion);

string newTemp();

E\_place genExpression(TreeNode \*node);

///@return value has't been decided.

void genAssignment(TreeNode \*node);

T\_place genTerm(TreeNode \*node);

F\_place genFactor(TreeNode \*node);

codeIndex genCondition(TreeNode \*node);

codeIndex genSelectionStatement(TreeNode \*node);

codeIndex genStatement(TreeNode \*node);

codeIndex genStatements(TreeNode \*node);

codeIndex genWhileStatement(TreeNode \*node);

codeIndex genDoStatement(TreeNode \*node);

///@brief statement chain

codeIndex genStatementBlock(TreeNode \*node);

void genProgress(TreeNode \*node);

// void genComment(const string& info);

void showQuaternionList();

};

#endif //COMPILER\_INTERCODE\_H

//

// Created by 王若璇 on 16/12/22.

//

#include "InterCode.h"

int Quaternion::tolNum = 0;

int InterCode::tolTempNum = 0;

int Quaternion::tolLabel = 0;

InterCode::InterCode() {

Quaternion::tolNum = 0;

InterCode::tolTempNum = 0;

Quaternion::tolLabel = 0;

}

int Quaternion::getIndex\_() const {

return index\_;

}

void Quaternion::setIndex\_(int index\_) {

Quaternion::index\_ = index\_;

}

///@brief Expression intermediate Code.

InterCode::E\_place InterCode::genExpression(TreeNode \*node) {

if (node == nullptr || node->getChildren\_().empty()) {

return "none";

}

E\_place e1;

E\_place e2;

string res;

///@note auto pointer don't support inc\_op "++".otherwise it will lead to memory error.

for (int i = 0; i < node->getChildren\_().size(); i++) {

if (i == 0) {

e1 = genTerm(node->getChildren\_()[0]);

res = e1;

} else {

TreeNode \*it = node->getChildren\_()[i];

if (it->kind == nodeType::ADD || it->kind == nodeType::SUB) {

string tmpOp = it->kind == nodeType::ADD ? "+" : "-";

i++;

e2 = genTerm(node->getChildren\_()[i]);

res = newTemp();

// cerr<<"res "<<res<<endl;

emit(Quaternion(tmpOp, e1, e2, res));

// cout<<quaternionList\_.back().toString()<<endl;

e1 = res;

}

}

}

// cout<<res<<"res"<<endl;

return res;

}

void InterCode::emit(const Quaternion &quaternion) {

///later use fout.

#ifdef SHOW\_ROW\_NUM

// cout<<quaternion.row<<" :";

#endif

quaternionList\_.push\_back(quaternion);

// cout<<quaternion.toString()<<endl;

}

InterCode::T\_place InterCode::genTerm(TreeNode \*node) {

if (node == nullptr || node->getChildren\_().empty()) {

return "none";

}

T\_place t1;

T\_place t2;

string res;

// node->showNode();

// cout<<node->token.lexeme<<endl;

// cerr<<node->kind<<" " <<node->getChildren\_().size()<<endl;

for (int i = 0; i < node->getChildren\_().size(); i++) {

if (i == 0) {

t1 = genFactor(node->getChildren\_()[0]);

res = t1;

} else {

TreeNode \*it = node->getChildren\_()[i];

if (it->kind == nodeType::MULT || it->kind == nodeType::DIV) {

string tmpOp = it->kind == nodeType::MULT ? "\*" : "/";

i++;

TreeNode \*next = node->getChildren\_()[i];

t2 = genFactor(next);

res = newTemp();

emit(Quaternion(tmpOp, t1, t2, res));

t1 = res;

}

}

}

return res;

}

InterCode::F\_place InterCode::genFactor(TreeNode \*node) {

if (node->kind == nodeType::EXPRESSION) {

return genExpression(node);

} else if (node->kind == nodeType::ID) {

return node->token.lexeme;

} else if (node->kind == nodeType::BINARY\_NUMBER) {

if (node->token.empty()) {

return "num";

} else {

return node->token.lexeme;

}

} else {

Error::syntaxError("empty Factor!");

}

}

string InterCode::newTemp() {

///push it to symbolTable.

return "t" + to\_string(tolTempNum++);

}

const vector<Quaternion> &InterCode::getQuaternionList\_() const {

return quaternionList\_;

}

void InterCode::setQuaternionList\_(const vector<Quaternion> &quaternionList\_) {

InterCode::quaternionList\_ = quaternionList\_;

}

void InterCode::emit(const string &str) {

// cout<<str<<endl;

}

void InterCode::genAssignment(TreeNode \*node) {

///@brief according to the structure of AST.

emit(Quaternion("=", genExpression(node->getChildren\_()[2]), "", node->getChildren\_()[0]->token.lexeme));

}

///use pointer is smarter.

///@todo to support "||" and "&&" operation.

InterCode::codeIndex InterCode::genCondition(TreeNode \*node) {

string arg1;

if (node->kind != nodeType::CONDITION) {

Error::syntaxError("irregular condition!");

}

//assert(condition.row==Quaternion::tolNum);

if (node->getChildren\_().size() == 1) {

///single expression

arg1 = genExpression(node->getChildren\_()[0]);

} else if (node->getChildren\_().size() == 3) {

///binary expression

arg1 = genExpression(node->getChildren\_()[0]) + node->getChildren\_()[1]->token.lexeme +

genExpression(node->getChildren\_()[2]);

//this->showQuaternionList();

} else;///irregular condition.

///@note ensure the correct index.

///@bug fix it.

Quaternion condition;

condition.arg1 = arg1;

condition.trueList = Quaternion::tolNum;

condition.row = Quaternion::tolNum;

condition.falseList = condition.row + 1;

condition.beginCode = condition.row;

condition.isCondition = true;

Quaternion gotoZero;

gotoZero.isJump = true;

emit(condition);

emit(gotoZero);

return condition.getIndex\_();

}

InterCode::codeIndex InterCode::genSelectionStatement(TreeNode \*node) {

if (node->kind != nodeType::SELECTION\_STATEMENT) {

Error::syntaxError("irregular selection\_statement");

}

int index = genCondition(node->getChildren\_()[0]);

///trueList

// cout<<"l"<<Quaternion::tolNum<<endl;

bathPatch\_(index, Quaternion::tolNum + 1);

///C\_chain is the index in the list.

int C\_chain = quaternionList\_[index].falseList - 1;

///S1\_chain is the head of list.

int S1\_chain = genStatementBlock(node->getChildren\_()[1]);

if (node->getChildren\_().size() == 2) {

///non-else statement

///@note a little different

///failList

merge\_(C\_chain, Quaternion::tolNum + 1);

} else if (node->getChildren\_().size() == 3) {

int tmpFailList = Quaternion::tolNum; ///tmpQ index

Quaternion tmpQ;

tmpQ.isJump = true;

emit(tmpQ);

///the same

bathPatch\_(C\_chain, Quaternion::tolNum + 1);

///S2\_chain is the head of list

int S2\_chain = genStatementBlock(node->getChildren\_()[2]);

int outIndex = Quaternion::tolNum + 1;

merge\_(tmpFailList, outIndex);

} else {

Error::syntaxError("interCode::irregular selection statement");

}

return 0;

}

void InterCode::bathPatch\_(InterCode::codeIndex index, int pos) {

quaternionList\_[index].result = to\_string(pos);

quaternionList\_[index].assigned = true;

// cout<<pos<<" "< <quaternionList\_[index].toString()<<endl;

}

InterCode::codeIndex InterCode::genStatementBlock(TreeNode \*node) {

if (node->kind != nodeType::STATEMENT\_BLOCK) {

Error::syntaxError("interCode::irregular selection statement");

}

if (node->getChildren\_().size() == 1) {

genStatements(node->getChildren\_()[0]);

}

return Quaternion::tolNum;

}

///@note

InterCode::codeIndex InterCode::merge\_(InterCode::codeIndex p1, InterCode::codeIndex p2) {

quaternionList\_[p1].result = to\_string(p2);

quaternionList\_[p1].assigned = true;

return 0;

}

void InterCode::showQuaternionList() {

for (auto it:quaternionList\_) {

#ifdef SHOW\_ROW\_NUM

cout << it.row << ": ";

#endif

cout << it.toString() << endl;

}

}

InterCode::codeIndex InterCode::genStatement(TreeNode \*node) {

if (node->kind == nodeType::ASSIGN\_STATEMENT) {

genAssignment(node);

} else if (node->kind == nodeType::SELECTION\_STATEMENT) {

genSelectionStatement(node);

} else if (node->kind == nodeType::DO\_WHILE\_STATEMENT) {

} else if (node->kind == nodeType::WHILE\_STATEMENT) {

genWhileStatement(node);

} else;

return 0;

}

InterCode::codeIndex InterCode::genStatements(TreeNode \*node) {

if (node->kind != nodeType::STATEMENTS) {

Error::syntaxError("interCode::irregular statements");

}

for (auto it:node->getChildren\_()) {

genStatement(it);

}

return 0;

}

InterCode::codeIndex InterCode::genWhileStatement(TreeNode \*node) {

if (node->kind == nodeType::WHILE\_STATEMENT) {

int index = genCondition(node->getChildren\_()[0]);

bathPatch\_(index, Quaternion::tolNum + 1);

int W\_chain = quaternionList\_[index].falseList - 1;

int S\_chain = genStatementBlock(node->getChildren\_()[1]);

merge\_(W\_chain, Quaternion::tolNum + 1);

Quaternion tmpQ;

tmpQ.isJump = true;

tmpQ.assigned = true;

tmpQ.result = to\_string(index + 1);

emit(tmpQ);

}

return Quaternion::tolNum;

}

InterCode::codeIndex InterCode::genDoStatement(TreeNode \*node) {

return 0;

}

void InterCode::genProgress(TreeNode \*node) {

if (node->kind == nodeType::MAIN && node->getChildren\_().size() == 1) {

genStatementBlock(node->getChildren\_()[0]);

} else {

Error::syntaxError("interCode:: irregular progress");

}

emit(Quaternion("", "", "", "END"));

}

//

// Created by 王若璇 on 16/10/26.

//

/\*

\* Scanner 类用来提取源代码中的token

\* 没有实现检测是否合法的功能

\* \*/

#ifndef SEEDCUP\_SCANNER\_H

#define SEEDCUP\_SCANNER\_H

//#define SHOW\_FIN

//#define DEBUG\_TOKEN

#include <string>

#include <fstream>

#include <deque>

#include <set>

#include <iostream>

#include <cassert>

#include <map>

#include "Error.h"

using namespace std;

///@brief Scanenr class to get token from source file.

class Scanner {

private:

const static unsigned INFROW = 0x3f3f3f3f;

enum State { //各种状态

START\_STATE,

INT\_STATE,

ID\_STATE,

SYMBOL\_STATE,

CHAR\_STATE,

STRING\_STATE,

ESCAPE\_STRING\_STATE,

LINE\_COMMENT\_STATE,

BLOCK\_COMMENT\_STATE,

PREEND\_BLOCK\_COMMENT\_STATE,

DONE\_STATE,

ERROR\_STATE,

DOUBLE\_STATE

};

public:

set<string> keyWords;

set<string> symbols;

///@note explicit define the enum type value.

enum TokenType {

ENDOFFILE = 100,

///@brief terminal symbol

///key word

MAIN = 1,

INT = 2,

FLOAT = 3,

DOUBLE = 4,

CHAR = 5,

IF = 6,

ELSE = 7,

DO = 8,

WHILE = 9,

///variable

STRING = 10,

ID = 10,

BINARY\_NUMBER = 20,

DOUBLE\_NUMBER = 20,

INT\_NUMBER = 20,

///symboL

ADD = 22,

ASSIGN = 21,

SUB = 23,

MULT = 24,

DIV = 25,

LEFT\_PAR = 26,

RIGHT\_PAR = 27,

LEFT\_BLOCK = 28,

RIGHT\_BLOCK = 29,

COMMA = 30,

SEMICOLON = 31,

GREATER = 32,

GREATER\_EQUAL = 33,

SMALLER = 34,

SMALLER\_EQUAL = 35,

EQUAL = 36,

NON\_EQUAL = 37,

HASH\_MARK = 38,

BOOL = 39,

///@brief TOP symbol

KEY\_WORD = 50,

SYMBOL = 51,

NONE = 52,

ERROR = 53,

END\_MARK = 100

///@todo ( +|-|ε ) dd\*(.dd\* | ε)( e ( +|-|ε ) dd\*|ε)

};

struct Token {

TokenType kind; //token 的类型

string lexeme; //token 的值

unsigned int row; //当前的行数

unsigned int pos; //当前行的位置

Token() {

kind = NONE;

lexeme.clear();

row = INFROW;

}

bool empty() {

return kind == NONE && lexeme.empty();;

}

};

void initKeyWords();

void initSymbols();

private:

string lineBuffer; //缓冲行，保存程序的某一行数据

unsigned int bufferPos; //缓冲行的索引

unsigned int row; //当前的行号

ifstream fin;

Token pre;

map<string, TokenType> keyMap;

private:

void initKeyMap();

char nextChar();

void rollBack();

TokenType searchReserved(const string &s);

void setTokenType(Token &rhs, TokenType nowType, unsigned int nowRow, unsigned int nowPos);

void fixTokenType(Token &rhs);

public:

Scanner();

void openFile(const string &filename);

void closeFile();

Token nextToken();

unsigned int getRow();

string getLineBuffer();

void resetRow();

void cleanUp();

#ifdef SHOW\_FIN

void showFin();

#endif

};

#endif //SEEDCUP\_SCANNER\_H

//

// Created by 王若璇 on 16/10/26.

//

#include "Scanner.h"

void Scanner::initKeyWords() {

/\*\*

\* @note to use initialize\_list

keyWords.clear();

keyWords.insert("int");

keyWords.insert("printf");

keyWords.insert("void");

keyWords.insert("if");

keyWords.insert("else");

keyWords.insert("else if");

keyWords.insert("return");

keyWords.insert("while");

keyWords.insert("break");

keyWords.insert("do");

keyWords.insert("for");

\* \*/

keyWords = {{"int"},

{"printf"},

{"void"},

{"if"},

{"else"},

{"else"},

{"else if"},

{"return"},

{"while"},

{"break"},

{"do"},

{"for"},

{"main"}

};

}

void Scanner::initSymbols() {

symbols.clear();

symbols.insert("{");

symbols.insert("}");

symbols.insert("(");

symbols.insert(")");

symbols.insert("[");

symbols.insert("]");

symbols.insert(";");

symbols.insert(".");

symbols.insert(",");

symbols.insert("+");

symbols.insert("-");

symbols.insert("\*");

symbols.insert("/");

symbols.insert(">");

symbols.insert(">=");

symbols.insert("<");

symbols.insert("<=");

symbols.insert("=");

symbols.insert("==");

symbols.insert("!=");

symbols.insert("/\*");

symbols.insert("\*/");

symbols.insert("//");

symbols.insert("++");

symbols.insert("--");

symbols.insert("|");

symbols.insert("^");

symbols.insert("~");

symbols.insert("!");

symbols.insert("&&");

symbols.insert("||");

symbols.insert("#");

}

char Scanner::nextChar() {

if (bufferPos >= lineBuffer.length()) {

row++;

getline(fin, lineBuffer);

lineBuffer += '\n';

//bufferPos = 0;

//cerr<<"row "<<row<<" "<<lineBuffer.size()<<endl;

#ifdef DEBUG\_TOKEN

cerr<<"row "<<row<<" "<<lineBuffer<<endl;

#endif

if (!fin.fail()) {

bufferPos = 0;

return lineBuffer[bufferPos++];

} else {

//cerr<<"ok"<<endl;

return EOF;

}

} else {

return lineBuffer[bufferPos++];

}

}

void Scanner::rollBack() {

assert(bufferPos > 0);

bufferPos--;

}

Scanner::TokenType Scanner::searchReserved(const string &s) {

if (keyWords.find(s) != keyWords.end()) {

return KEY\_WORD;

} else {

return ID;

}

}

void Scanner::openFile(const string &filename) { //在这个地方注意exit 和 return的区别。

if (filename.empty()) {

std::cerr << "filename is Empty!" << endl;

return;

}

string suffix\_txt(filename.substr(filename.length() - 4, 4));;

string suffix\_c(filename.substr(filename.length() - 2, 2));

if (suffix\_c == ".c" || suffix\_txt == ".txt") {

fin.open(filename);

if (fin.fail() || !fin.is\_open()) {

cerr << "file " << filename << " does not exit" << endl;

return;

}

} else {

std::cerr << "Error :" << filename << " no a valid filename" << endl;

}

}

Scanner::Scanner() {

row = 0;

bufferPos = 0;

initKeyWords();

initSymbols();

initKeyMap();

}

void Scanner::closeFile() {

fin.close();

}

Scanner::Token Scanner::nextToken() {

Token resToken;

resToken.lexeme.clear();

State nowState = START\_STATE;

while (nowState != DONE\_STATE) {

char nowChar = nextChar();

//cout<<nowChar;

#ifdef DEBUG\_TOKEN

cout<<nowChar<<" --- "<<endl;

#endif

if (nowChar == EOF) {

//cout<<"ok"<<endl;

resToken.kind = ENDOFFILE;

break;

}

//cout<<resToken.lexeme<<" ";

switch (nowState) {

case START\_STATE:

if (nowChar == '\t' || nowChar == '\n' || nowChar == ' ' || nowChar == '\r' || nowChar == '\b') {

} else if (isalpha(nowChar) || nowChar == '\_') { //处理标识符和关键字

nowState = ID\_STATE;

resToken.lexeme += nowChar;

setTokenType(resToken, ID, row, bufferPos);

} else if (isdigit(nowChar)) { //处理整型数字

nowState = INT\_STATE;

resToken.lexeme += nowChar;

setTokenType(resToken, INT\_NUMBER, row, bufferPos);

} else if (symbols.find({nowChar}) != symbols.end()) {//处理符号

nowState = SYMBOL\_STATE;

resToken.lexeme += nowChar;

setTokenType(resToken, SYMBOL, row, bufferPos);

} else if (nowChar == '"') {//处理字符串

///@bug conflict with ID\_STATE

nowState = STRING\_STATE;

setTokenType(resToken, STRING, row, bufferPos);

} else if (nowChar == '\'') { //处理单个字符

nowState = CHAR\_STATE;

setTokenType(resToken, CHAR, row, bufferPos);

} else {

nowState = ERROR\_STATE; //处理错误格式

setTokenType(resToken, ERROR, row, bufferPos);

resToken.lexeme += nowChar;

}

break;

case ID\_STATE:

//cout<<resToken.lexeme<<endl;

if (isdigit(nowChar) || isalpha(nowChar) || nowChar == '\_') {

resToken.lexeme += nowChar;

} else {

nowState = DONE\_STATE;

rollBack();

if (searchReserved(resToken.lexeme) == KEY\_WORD) {

resToken.kind = KEY\_WORD;

fixTokenType(resToken);

//cout<<resToken.lexeme<<" --"<<endl;

} else {

resToken.kind = ID;

}

}

break;

case INT\_STATE:

if (isdigit(nowChar)) {

resToken.lexeme += nowChar;

resToken.kind = INT\_NUMBER;

} else if (nowChar == '.' || nowChar == 'e') {

rollBack();

//resToken.lexeme += nowChar;

nowState = DOUBLE\_STATE;

} else {

rollBack();

resToken.kind = Scanner::INT\_NUMBER;

nowState = DONE\_STATE;

}

break;

case STRING\_STATE:

if (nowChar == '"') {

nowState = DONE\_STATE;

} else if (nowChar == '\\') { //转义部分有问题再考虑

nowState = ESCAPE\_STRING\_STATE;

resToken.lexeme += nowChar;

} else {

resToken.lexeme += nowChar;

}

break;

//switch 位置顺序不同 编译不通过？

case ESCAPE\_STRING\_STATE:

nowState = STRING\_STATE;

resToken.lexeme.pop\_back();

resToken.lexeme += nowChar;

break;

case LINE\_COMMENT\_STATE:

if (nowChar == '\n') {

nowState = DONE\_STATE;

}

break;

case BLOCK\_COMMENT\_STATE:

if (nowChar == '\*') {

nowState = PREEND\_BLOCK\_COMMENT\_STATE;

resToken.lexeme += nowChar;

}

break;

case PREEND\_BLOCK\_COMMENT\_STATE:

if (nowChar == '/') {

nowState = DONE\_STATE;

resToken.lexeme += nowChar;

}

break;

//

case DOUBLE\_STATE:

setTokenType(resToken, DOUBLE\_NUMBER, row, bufferPos);

if (nowChar == '.') {

resToken.lexeme += nowChar;

nowChar = nextChar();

if (!isdigit(nowChar)) {

Error::numberError("number format error");

}

while (isdigit(nowChar)) {

resToken.lexeme += nowChar;

nowChar = nextChar();

}

}

// cerr<<" now "<<nowChar<<" "<<resToken.lexeme<<endl;

if (nowChar == 'e') {

resToken.lexeme += nowChar;

nowChar = nextChar();

if (nowChar == '+' || nowChar == '-') {

resToken.lexeme += nowChar;

nowChar = nextChar();

}

if (!isdigit(nowChar)) {

Error::numberError("number format error");

}

while (isdigit(nowChar)) {

resToken.lexeme += nowChar;

nowChar = nextChar();

}

rollBack();

} else {

//cout<<"ok"<<endl;

rollBack();

}

nowState = DONE\_STATE;

break;

case SYMBOL\_STATE: //符号分为单符号和双符号

string pre = resToken.lexeme;

/\*处理其它的符号\*/

if (nowChar == '=') {

if (!pre.empty()) { //双符号

resToken.lexeme += nowChar;

nowState = DONE\_STATE;

}

} else if (nowChar == '/' && pre == "/") {

nowState = LINE\_COMMENT\_STATE;

resToken.lexeme += nowChar;

} else if (nowChar == '\*' && pre == "/") {

nowState = BLOCK\_COMMENT\_STATE;

resToken.lexeme += nowChar;

} else if (nowChar == '|' && pre == "|") {

nowState = DONE\_STATE;

resToken.lexeme += nowChar;

} else if (nowChar == '&' && pre == "&") {

nowState = DONE\_STATE;

resToken.lexeme += nowChar;

} else if (pre == "+" || pre == "-") {

if (isdigit(nowChar)) {

//cout<<tmp<<endl;

rollBack();

nowState = INT\_STATE;

} else {

rollBack();

nowState = DONE\_STATE;

}

} else {

rollBack();

nowState = DONE\_STATE;

}

fixTokenType(resToken);

break;

/\*@delete

\* else if (nowChar == '+' && pre == "+") {

nowState = DONE\_STATE;

resToken.lexeme += nowChar;

} else if (nowChar == '-' && pre == "-") {

nowState = DONE\_STATE;

resToken.lexeme += nowChar;

} \*/

}

}

//cerr<<lineBuffer<<" "<<resToken.lexeme<<endl;

fixTokenType(resToken);

return resToken;

}

void Scanner::resetRow() {

row = 0;

}

void Scanner::setTokenType(Scanner::Token &rhs, Scanner::TokenType nowType, unsigned int nowRow, unsigned int nowPos) {

rhs.pos = nowPos;

rhs.kind = nowType;

rhs.row = nowRow;

}

unsigned int Scanner::getRow() {

return row;

}

string Scanner::getLineBuffer() {

return lineBuffer;

}

void Scanner::initKeyMap() {

keyMap = {

{"main", MAIN},

{"int", INT},

{"float", FLOAT},

{"double", DOUBLE},

{"char", CHAR},

{"if", IF},

{"else", ELSE},

{"do", DO},

{"while", WHILE},

{"l(l|d)\*", STRING},

{"( +|-|ε ) dd\*(.dd\* | ε)( e ( +|-|ε ) dd\*|ε) ", BINARY\_NUMBER},

{"=", ASSIGN},

{"+", ADD},

{"-", SUB},

{"\*", MULT},

{"/", DIV},

{"(", LEFT\_PAR},

{")", RIGHT\_BLOCK},

{"{", LEFT\_BLOCK},

{"}", RIGHT\_BLOCK},

{",", COMMA},

{";", SEMICOLON},

{">", GREATER},

{">=", GREATER\_EQUAL},

{"<", SMALLER},

{"<=", SMALLER\_EQUAL},

{"==", EQUAL},

{"!=", NON\_EQUAL},

{"#", HASH\_MARK}

};

}

void Scanner::cleanUp() {

bufferPos = 0;

row = 0;

closeFile();

///@todo try to use try catch to solve exception.

}

void Scanner::fixTokenType(Scanner::Token &rhs) {

switch (rhs.kind) {

case KEY\_WORD:

if (rhs.lexeme == "main") {

rhs.kind = MAIN;

} else if (rhs.lexeme == "int") {

rhs.kind = INT;

} else if (rhs.lexeme == "double") {

rhs.kind = DOUBLE;

} else if (rhs.lexeme == "float") {

rhs.kind = FLOAT;

} else if (rhs.lexeme == "char") {

rhs.kind = CHAR;

} else if (rhs.lexeme == "if") {

rhs.kind = IF;

} else if (rhs.lexeme == "else") {

rhs.kind = ELSE;

} else if (rhs.lexeme == "do") {

rhs.kind = DO;

} else if (rhs.lexeme == "while") {

rhs.kind = WHILE;

} else;

break;

case SYMBOL:

if (rhs.lexeme == "+") {

rhs.kind = ADD;

} else if (rhs.lexeme == "-") {

rhs.kind = SUB;

} else if (rhs.lexeme == "\*") {

rhs.kind = MULT;

} else if (rhs.lexeme == "/") {

rhs.kind = DIV;

} else if (rhs.lexeme == "(") {

rhs.kind = LEFT\_PAR;

} else if (rhs.lexeme == ")") {

rhs.kind = RIGHT\_PAR;

} else if (rhs.lexeme == "{") {

rhs.kind = LEFT\_BLOCK;

} else if (rhs.lexeme == "}") {

rhs.kind = RIGHT\_BLOCK;

} else if (rhs.lexeme == ",") {

rhs.lexeme = COMMA;

} else if (rhs.lexeme == ";") {

rhs.kind = SEMICOLON;

} else if (rhs.lexeme == ">") {

rhs.kind = GREATER;

} else if (rhs.lexeme == ">=") {

rhs.kind = GREATER\_EQUAL;

} else if (rhs.lexeme == "<") {

rhs.kind = SMALLER;

} else if (rhs.lexeme == "<=") {

rhs.kind = SMALLER\_EQUAL;

} else if (rhs.lexeme == "==") {

rhs.kind = EQUAL;

} else if (rhs.lexeme == "=") {

rhs.kind = ASSIGN;

} else if (rhs.lexeme == "!=") {

rhs.kind = NON\_EQUAL;

} else if (rhs.lexeme == "#") {

rhs.kind = HASH\_MARK;

}

break;

default:

break;

}

}

#ifdef SHOW\_FIN

void Scanner::showFin() {

while (!fin.eof()){

string str;

getline(fin,str);

cout<<str<<endl;

}

}

#endif

//

// Created by 王若璇 on 16/12/14.

//

#ifndef COMPILER\_RECDESPARSER\_H

#define COMPILER\_RECDESPARSER\_H

#include "Parser.h"

#include "Error.h"

//#include "Util.h"

#include "AST.h"

#include <string>

#include <sstream>

#include "Scanner.h"

///@brief RecDesParser class use Strategy pattern implement.

///public inheritance.

class RecDesParser : public Parser {

using tokenType = Scanner::TokenType;

using nodeType = TreeNode::NodeType;

public:

void recusiveDescentParse();

TreeNode \*progress();

///statement

TreeNode \*statementBlock();

TreeNode \*statements();

TreeNode \*statement();

TreeNode \*assignStatement();

TreeNode \*loopStatement();

TreeNode \*do\_whileStatement();

TreeNode \*whileStatement();

TreeNode \*parCondition();

TreeNode \*elseStatement();

TreeNode \*selectionStatement();

TreeNode \*condition();

///expression

TreeNode \*expression();

TreeNode \*term();

TreeNode \*factor();

///other

TreeNode \*num();

TreeNode \*id();

TreeNode \*relationalOp();

};

#endif //COMPILER\_RECDESPARSER\_H

//

// Created by 王若璇 on 16/12/14.

//

#include "RecDesParser.h"

///@todo can use c++ design pattern Strategy pattern.

/\*\*

\* symbol: < x > x appear once

\* { x } x appear more than zero

\* [ x ] x appear once or not appear

\* \*/

pair<bool, int> mystringToInt(const string &str) noexcept(false) {

int res;

istringstream istr{str};

// throw "exp";

if (!(istr >> res)) {

assert(0);

throw "type conversion failed";

}

return make\_pair(istr.eof(), res);

}

pair<bool, double> mystringToDouble(const string &str) noexcept(false) {

double res;

istringstream istr{str};

if (!(istr >> res)) {

throw "type conversion failed";

}

return make\_pair(istr.eof(), res);

}

void RecDesParser::recusiveDescentParse() {

this->ast.setRoot(progress());

}

///@brief Progress ::= main()<statementBlock>

TreeNode \*RecDesParser::progress() {

TreeNode \*node = new TreeNode;

///set the node type.

Scanner::Token token = this->nextToken();

// cout<<token.lexeme<<token.kind<<endl;

if (token.kind != Scanner::TokenType::MAIN) {

///solve the syntax error.

Error::exceptSyntax(token.lexeme, "main");

} else {

if (this->nextToken().kind == tokenType::LEFT\_PAR &&

this->nextToken().kind == tokenType::RIGHT\_PAR) {

///ensure the syntax correct.

node->terminal = true;

//node->key = token.lexeme;

node->kind = nodeType::MAIN;

node->token = token;

node->addChild(statementBlock());

} else {

///solve the syntax error.

Error::syntaxError("symbol format error!");

}

}

return node;

}

///@brief statementBlock ::= '{' <statements> '}'

TreeNode \*RecDesParser::statementBlock() {

TreeNode \*node = new TreeNode;

Scanner::Token token = this->nextToken();

if (token.kind == tokenType::LEFT\_BLOCK) {

///ensure the syntax correct.

node->kind = nodeType::STATEMENT\_BLOCK;

node->addChild(statements());

// {

// cout<<"\*\* "<<this->nextToken().lexeme<<endl;

// rollbackToken();

// }

if (this->nextToken().kind != tokenType::RIGHT\_BLOCK) {

///right parenthesis missing

cerr<<token.row<<" row"<<endl;

Error::exceptSyntax("missing right block", "}");

}

}

return node;

}

///@brief statements ::= statement { ;<statement> };

TreeNode \*RecDesParser::statements() {

TreeNode \*node = new TreeNode;

TreeNode \*child = nullptr;

bool flag = false;

do {

child = statement();

if (child != nullptr) {

flag = true;

node->addChild(child);

//child = nullptr;

if (this->nextToken().kind != tokenType::SEMICOLON) {

cerr<<this->nextToken().row<<" row"<<endl;

rollbackToken();

Error::exceptSyntax("missing semicolon ';' ", " symbol ';' ");

}

}

} while (child != nullptr);

if (flag) {

node->kind = nodeType::STATEMENTS;

return node;

} else {

delete node;

return nullptr;

}

}

///@brief statement ::= assignStatement | selectionStatement | loopStatement

TreeNode \*RecDesParser::statement() {

TreeNode \*node = nullptr;

///look ahead one token.

Scanner::Token token = this->nextToken();

this->rollbackToken();

if (token.kind == tokenType::ID) {

node = assignStatement();

node->kind = nodeType::ASSIGN\_STATEMENT;

} else if (token.kind == tokenType::DO) {

node = do\_whileStatement();

node->kind = nodeType::DO\_WHILE\_STATEMENT;

} else if (token.kind == tokenType::IF) {

node = selectionStatement();

node->kind = nodeType::SELECTION\_STATEMENT;

} else if(token.kind == tokenType::WHILE){

node = whileStatement();

node->kind = nodeType::WHILE\_STATEMENT;

} else{

// cout<<token.lexeme<<endl;

// Error::syntaxError("empty statement!");

}

return node;

}

///@brief assignStatement ::= ID '=' <expression>

TreeNode \*RecDesParser::assignStatement() {

TreeNode \*node = new TreeNode;

node->kind = nodeType::ASSIGN\_STATEMENT;

node->addChild(id());

Scanner::Token token = this->nextToken();

if (token.kind != tokenType::ASSIGN) {

Error::syntaxError("irregular assignStatement");

}

TreeNode \*tmp = new TreeNode;

tmp->kind = nodeType::ASSIGN;

tmp->token = token;

tmp->terminal = true;

node->addChild(tmp);

node->addChild(expression());

return node;

}

///@brief loopStatement ::= do\_whileStatement | whileStatement | forStatement

///@note more strong function.

TreeNode \*RecDesParser::loopStatement() {

TreeNode \*node = nullptr;

Scanner::Token token = this->nextToken();

rollbackToken();

if (token.kind == tokenType::DO) {

node = do\_whileStatement();

} else if (token.kind == tokenType::WHILE) {

node = whileStatement();

} else if (/\*(token.kind==tokenType::FOR\*/true) {

// node = forStatement();

} else {

Error::syntaxError("irregular loopStatement");

}

return node;

}

///@brief do\_whileStatement ::= do <statementBlock> while ( <condition> )

TreeNode \*RecDesParser::do\_whileStatement() {

TreeNode \*node;

if (nextToken().kind == tokenType::DO) {

node = new TreeNode;

node->kind = nodeType::DO\_WHILE\_STATEMENT;

node->addChild(statementBlock());

if (nextToken().kind == tokenType::WHILE) {

node->addChild(parCondition());

} else {

rollbackToken();

Error::exceptSyntax("missing while\_condition", "while");

}

} else {

Error::syntaxError("empty do\_whileStatement");

}

return node;

}

///@brief whileStatement ::= while ( <condition> ) <statementBlock>

TreeNode \*RecDesParser::whileStatement() {

TreeNode \*node;

if(nextToken().kind==tokenType::WHILE){

node = new TreeNode;

node->kind = nodeType::WHILE\_STATEMENT;

node->addChild(parCondition());

node->addChild(statementBlock());

} else{

rollbackToken();

Error::exceptSyntax("missing key\_word ","while");

}

return node;

}

///@brief selectionStatement ::= if ( <condition> ) <statementBlock> [else <statementBlock>]

TreeNode \*RecDesParser::selectionStatement() {

TreeNode \*node = new TreeNode;

node->kind = nodeType::SELECTION\_STATEMENT;

if (nextToken().kind == tokenType::IF) {

//cout<<"if"<<endl;

node->addChild(parCondition());

node->addChild(statementBlock());

} else {

rollbackToken();

Error::syntaxError("empty if condition");

}

if (nextToken().kind == tokenType::ELSE) {

node->addChild(statementBlock());

} else {

rollbackToken();

}

return node;

}

TreeNode \*RecDesParser::parCondition() {

TreeNode \*node = nullptr;

if (nextToken().kind == tokenType::LEFT\_PAR) {

node = condition();

if (nextToken().kind != tokenType::RIGHT\_PAR) {

rollbackToken();

Error::exceptSyntax("missing right parenthesis", ")");

}

} else {

///rollback twice

rollbackToken();

Error::syntaxError("irregular par\_condition");

}

return node;

}

TreeNode \*RecDesParser::elseStatement() {

return nullptr;

}

///@brief condition ::= <expression> <relationalOp> <expression>

TreeNode \*RecDesParser::condition() {

TreeNode \*node = new TreeNode;

node->kind = nodeType::CONDITION;

node->addChild(expression());

node->addChild(relationalOp());

node->addChild(expression());

return node;

}

///@brief expression ::= <term> {+ <term> | - <term>}

TreeNode \*RecDesParser::expression() {

TreeNode \*node = new TreeNode;

node->kind = nodeType::EXPRESSION;

TreeNode \*child = term();

node->addChild(child);

Scanner::Token token;

while (!(token = this->nextToken()).empty() && (token.kind == tokenType::ADD || token.kind == tokenType::SUB)) {

TreeNode \*tmp = new TreeNode;

tmp->kind = token.kind == tokenType::ADD ? nodeType::ADD : nodeType::SUB;

///@note not contain more detail about token.

tmp->token = token;

tmp->terminal = true;

node->addChild(tmp);

node->addChild(term());

}

rollbackToken();

return node;

}

///@brief term ::= <factor> {\* <factor> | / <factor>}

TreeNode \*RecDesParser::term() {

TreeNode \*node = new TreeNode;

node->kind = nodeType::TERM;

TreeNode \*child = factor();

node->addChild(child);

Scanner::Token token;

while (!(token = nextToken()).empty() && (token.kind == tokenType::MULT || token.kind == tokenType::DIV)) {

TreeNode \*tmp = new TreeNode;

tmp->kind = token.kind == tokenType::MULT ? nodeType::MULT : nodeType::DIV;

tmp->token = token;

tmp->terminal = true;

node->addChild(tmp);

node->addChild(factor());

}

rollbackToken();

return node;

}

///@brief factor ::= ID | NUM | (expression)

TreeNode \*RecDesParser::factor() {

TreeNode \*node = nullptr;

///look ahead one token.

Scanner::Token token = this->nextToken();

this->rollbackToken();

if (token.kind == tokenType::ID) {

node = id();

} else if (token.kind == tokenType::BINARY\_NUMBER) {

node = num();

} else if (token.kind == tokenType::LEFT\_PAR) {

///ignore the left parenthesis.

this->nextToken();

node = expression();

///solve syntax error.

if (this->nextToken().kind != tokenType::RIGHT\_PAR) {

Error::syntaxError("irregular expression miss right parenthesis.");

}

}

return node;

}

TreeNode \*RecDesParser::num() {

Scanner::Token token = this->nextToken();

TreeNode \*node = new TreeNode;

if (token.kind == tokenType::BINARY\_NUMBER) {

///this function may throw exception.

node->terminal = true;

node->token = token;

try {

pair<bool, double> tmpd = mystringToDouble(token.lexeme);

pair<bool, int> tmpi = mystringToInt(token.lexeme);

///store tmpd or tmpi in node.

if (tmpd.first) {

node->kind = nodeType::DOUBLE\_NUMBER;

// node->value.dvalue = tmpd.second;

} else if (tmpi.first) {

node->kind = nodeType::INT\_NUMBER;

} else {

Error::syntaxError("type conversion error !");

}

} catch (const char \*s) {

cerr << "exception: " << s << endl;

}

} else {

///rollback and syntax error.

this->rollbackToken();

Error::syntaxError("irregular number !");

}

return node;

}

TreeNode \*RecDesParser::id() {

TreeNode \*node = nullptr;

Scanner::Token token = this->nextToken();

if (token.kind == tokenType::ID) {

node = new TreeNode;

node->kind = nodeType::ID;

node->terminal = true;

node->token = token;

} else {

Error::syntaxError("no vivid id");

}

return node;

}

TreeNode \*RecDesParser::relationalOp() {

TreeNode \*node = new TreeNode;

Scanner::Token token = this->nextToken();

node->token = token;

if (token.kind == tokenType::SMALLER) {

node->kind = nodeType::SMALLER;

} else if (token.kind == tokenType::SMALLER\_EQUAL) {

node->kind = nodeType::SMALLER\_EQUAL;

} else if (token.kind == tokenType::GREATER) {

node->kind = nodeType::GREATER;

} else if (token.kind == tokenType::GREATER\_EQUAL) {

node->kind = nodeType::GREATER\_EQUAL;

} else if (token.kind == tokenType::EQUAL) {

node->kind = nodeType::EQUAL;

} else if (token.kind == tokenType::NON\_EQUAL) {

node->kind = nodeType::NON\_EQUAL;

} else {

delete node;

node = nullptr;

this->rollbackToken();

Error::syntaxError("irregular relational operator !");

}

return node;

}