词法分析、语法分析程序实验

一、实验目的：扩充已有的样例语言TINY，为扩展TINY语言TINY＋构造词法分析和语法分析程序，从而掌握词法分析和语法分析程序的构造方法

二、实验内容：了解样例语言TINY及TINY编译器的实现，了解扩展TINY语言TINY＋，用EBNF描述TINY＋的语法，用C语言扩展TINY的词法分析和语法分析程序，构造TINY＋的递归下降语法分析器。

三、实验要求：将TINY＋源程序翻译成对应的TOKEN序列，并能检查一定的词法错误。将TOKEN序列转换成语法分析树，并能检查一定的语法错误。

四、样例语言TINY简介

T I N Y 的程序结构很简单，它在语法上与 A d a 或 P a s c a l 的 语 法 相 似 : 仅 是 一 个 由 分 号 分 隔 开 的语句序列。另外，它既无过程也无声明。所有的变量都是整型变量，通过对其赋值可较轻易 地声明变量(类似 FORTRAN或BASIC)。它只有两个控制语句: if语句和repeat语句，这两个 控制语句本身也可包含语句序列。 I f 语句有一个可选的 e l s e 部分且必须由关键字 e n d 结束。除此 之外， r e a d语句和 w r i t e 语句完成输入 / 输出。在花括号中可以有注释，但注释不能嵌套。

T I N Y 的表达式也局限于布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式 的比较组成，该比较使用 <与=比较算符。算术表达式可以包括整型常数、变量、参数以及 4个 整型算符+、-、\*、/，此外还有一般的数学属性。布尔表达式可能只作为测试出现在控制语 句中——而没有布尔型变量、赋值或 I / O 。

虽然 T I N Y 缺少真正程序设计语言所需要的许多特征：过程、数组且浮点值是一些较大的省略—但它足可以用来例证编译器的主要特征了。

Here is the definition for Tiny language

**The Tiny lexicon is as follows:**

* Keywords: **IF ELSE WRITE READ RETURN BEGIN END MAIN INT REAL**
* Single-character separators:   ;  ,  (   )
* Single-character operators:    +  -  \*   /
* Multi-character operators:    :=  ==   !=
* Identifier: An identifier consists of a letter followed by any number of letters or digits. The following are examples of identifiers: x, x2, xx2, x2x, End, END2.Note that **End** is an identifier while **END** is a keyword. The following are not identifiers:
  + IF, WRITE, READ, .... (keywords are not counted as identifiers)
  + 2x (identifier can not start with a digit)
  + Strings in comments are not identifiers.
* Number is a sequence of digits, or a sequence of digits followed by a dot, and followed by digits.
* Number -> Digits | Digits '.' Digits
* Digits -> Digit | Digit Digits

Digit  -> '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

* Comments: string between /\*\* and \*\*/. Comments can be longer than one line.

**The EBNF Grammar**

**High-level program structures**

Program -> MethodDecl MethodDecl\*

MethodDecl -> Type [MAIN] Id '(' FormalParams ')' Block

FormalParams -> [FormalParam ( ',' FormalParam )\* ]

FormalParam -> Type Id

Type -> INT | REAL

**Statements**

Block -> BEGIN Statement\* END

Statement -> Block

| LocalVarDecl

| AssignStmt

| ReturnStmt

| IfStmt

| WriteStmt

| ReadStmt

LocalVarDecl -> INT Id ';' | REAL Id ';'

AssignStmt -> Id := Expression ';'

ReturnStmt -> RETURN Expression ';'

IfStmt -> IF '(' BoolExpression ')' Statement

| IF '(' BoolExpression ')' Statement ELSE Statement

WriteStmt -> WRITE '(' Expression ',' QString ')' ';'

ReadStmt -> READ '(' Id ',' QString ')' ';'

QString is any sequence of characters except double quote itself, enclosed in double quotes.

**Expressions**

Expression -> MultiplicativeExpr (( '+' | '-' ) MultiplicativeExpr)\*

MultiplicativeExpr -> PrimaryExpr (( '\*' | '/' ) PrimaryExpr)\*

PrimaryExpr -> Num // Integer or Real numbers

| Id

| '(' Expression ')'

| Id '(' ActualParams ')'

BoolExpression -> Expression '==' Expression

|Expression '!=' Expression

ActualParams -> [Expression ( ',' Expression)\*]

**Sample program**

/\*\* this is a comment line in the sample program \*\*/

INT f2(INT x, INT y )

BEGIN

INT z;

z := x\*x - y\*y;

RETURN z;

END

INT MAIN f1()

BEGIN

INT x;

READ(x, "A41.input");

INT y;

READ(y, "A42.input");

INT z;

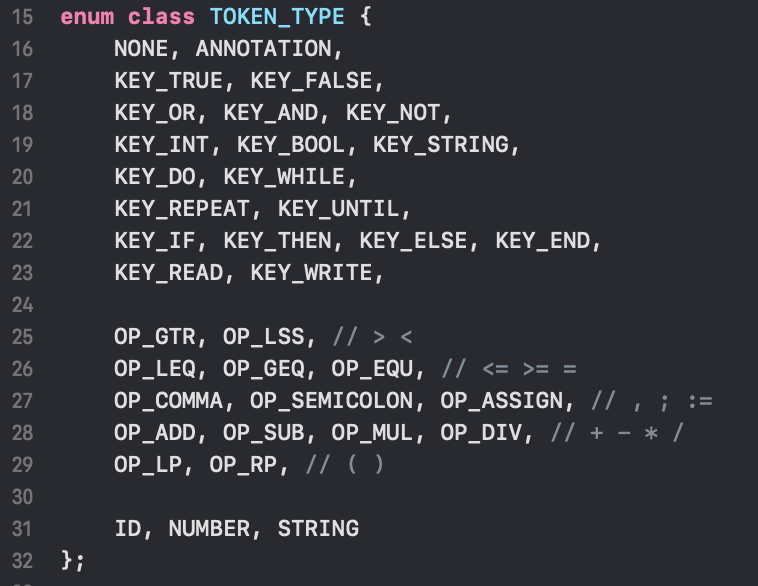
z := f2(x,y) + f2(y,x);

WRITE (z, "A4.output");

END

五、扩充内容

关键字如下：



六、词法分析器

**作用：**

滤掉源程序无用成分

处理与平台有关输入

根据模式识别记号，并交给语法分析器

调用符号表管理器或出错处理器，进行相关处理。

**工作方式：**

单独扫描，产生记号流供语法分析器使用。

作为语法分析器的子程序，并行工作。

**具体实现：**

class WordsAnalysis {

ifstream& in;

vector<Token> list;

private:

// temporary variables

string word;

TOKEN\_TYPE type;

int line\_number;

int char\_number;

bool isSign = false;

public:

WordsAnalysis(ifstream& fin) : in(fin) {

analyse();

}

vector<Token> getTokens() {

return list;

}

private:

void new\_line() {

line\_number ++;

char\_number = 0;

type = TOKEN\_TYPE::NONE;

isSign = false;

}

void close\_word() {

Token token;

token.token = word;

token.type = type;

TOKEN\_TYPE t = getType(word);

if (token.type == TOKEN\_TYPE::NONE)

token.type = t;

else if (token.type == TOKEN\_TYPE::ID && t != TOKEN\_TYPE::NONE)

token.type = t;

if (token.type != TOKEN\_TYPE::ANNOTATION)

list.push\_back(token);

word.clear();

type = TOKEN\_TYPE::NONE;

isSign = false;

}

void analyse() {

char ch;

line\_number = 1;

char\_number = 0;

isSign = false;

while (true)

{

in.get(ch);

char\_number ++;

if (in.fail() || in.eof()) {

if (type == TOKEN\_TYPE::STRING)

error("unclosed string", line\_number, char\_number);

if (type == TOKEN\_TYPE::ANNOTATION)

error("unclosed annotation", line\_number, char\_number);

if (!word.empty())

close\_word();

break;

}

if (isSign) {

if (getType(word) != TOKEN\_TYPE::NONE &&

getType(word + ch) == TOKEN\_TYPE::NONE)

close\_word();

else if (!isValidSign(ch))

close\_word();

}

if (!word.empty() && type != TOKEN\_TYPE::ANNOTATION && type != TOKEN\_TYPE::STRING

&& !isSign && isValidSign(ch))

close\_word();

if (word.empty()) {

if (ch == '{')

type = TOKEN\_TYPE::ANNOTATION;

else if (ch == '\'')

type = TOKEN\_TYPE::STRING;

else if (isDigit(ch))

type = TOKEN\_TYPE::NUMBER;

else if (isLetter(ch))

type = TOKEN\_TYPE::ID;

else if (isValidSign(ch))

isSign = true;

else if (!isSeparator(ch))

error(string("unknown symbol: ") + ch, line\_number, char\_number);

} else {

if (type == TOKEN\_TYPE::STRING && ch == '\'') {

word += ch;

close\_word();

continue;

}

if (type == TOKEN\_TYPE::ANNOTATION && ch == '}') {

word += ch;

close\_word();

continue;

}

}

if (ch == '\n') {

if (type == TOKEN\_TYPE::STRING) {

error("unclosed string", line\_number, char\_number);

close\_word();

new\_line();

continue;

}

if (type == TOKEN\_TYPE::ANNOTATION) {

error("unclosed annotation", line\_number, char\_number);

close\_word();

new\_line();

continue;

}

}

if (type != TOKEN\_TYPE::ANNOTATION && type != TOKEN\_TYPE::STRING) {

if (isSeparator(ch)) {

if (!word.empty())

close\_word();

if (ch == '\n')

new\_line();

continue;

}

}

if (type == TOKEN\_TYPE::NUMBER && !isDigit(ch))

error(string("here must be a digit: ") + ch, line\_number, char\_number);

if (type == TOKEN\_TYPE::ID && !isLetter(ch) && !isDigit(ch))

error(string("here must be a letter or digit: ") + ch, line\_number, char\_number);

word += ch;

if (ch == '\n')

new\_line();

}

if (!word.empty())

close\_word();

}

};

七、运行测试结果

---- START TEST test/test1.txt ----

ERROR IN LINE 10:1 unknown symbol: $

ERROR IN LINE 10:3 unknown symbol: &

ERROR IN LINE 11:4 here must be a digit: m

ERROR IN LINE 11:5 here must be a digit: a

ERROR IN LINE 11:6 here must be a digit: n

ERROR IN LINE 12:32 here must be a letter: '

ERROR IN LINE 13:34 unclosed string

ERROR IN LINE 14:24 here must be a letter: }

ERROR IN LINE 15:26 unclosed annotation

test failed.

------ END TEST test/test1.txt ----

---- START TEST test/test2.txt ----

(KEY\_INT, int)

(ID, A)

(OP\_COMMA, ,)

(ID, B)

(OP\_SEMICOLON, ;)

(KEY\_BOOL, bool)

(ID, C)

(OP\_SEMICOLON, ;)

(KEY\_STRING, string)

(ID, D)

(OP\_SEMICOLON, ;)

(ID, D)

(OP\_ASSIGN, :=)

(STRING, 'scanner')

(OP\_SEMICOLON, ;)

(ID, C)

(OP\_ASSIGN, :=)

(ID, A)

(KEY\_AND, and)

(KEY\_NOT, not)

(ID, B)

(OP\_SEMICOLON, ;)

(KEY\_WHILE, while)

(ID, A)

(OP\_LEQ, <=)

(ID, D)

(KEY\_DO, do)

(ID, A)

(OP\_ASSIGN, :=)

(ID, A)

(OP\_MUL, \*)

(NUMBER, 2)

(KEY\_END, end)

(OP\_SEMICOLON, ;)

(KEY\_IF, if)

(OP\_LP, ()

(ID, A)

(OP\_GTR, >)

(OP\_SUB, -)

(ID, B)

(OP\_RP, ))

(KEY\_THEN, then)

(ID, C)

(OP\_ASSIGN, :=)

(KEY\_TRUE, true)

(OP\_SEMICOLON, ;)

(KEY\_END, end)

(OP\_SEMICOLON, ;)

(KEY\_IF, if)

(OP\_LP, ()

(ID, A)

(OP\_LEQ, <=)

(OP\_SUB, -)

(ID, B)

(OP\_RP, ))

(KEY\_THEN, then)

(ID, C)

(OP\_ASSIGN, :=)

(KEY\_FALSE, false)

(OP\_SEMICOLON, ;)

(KEY\_END, end)

(OP\_SEMICOLON, ;)

test passed.

------ END TEST test/test2.txt ----

1 test(s) passed. 1 test(s) failed.