|  |
| --- |
| **计算机专业类课程** |
| **实验报告** |
| **课程名称：编译原理**  **学　　院：计算机科学与工程学院**  **专　　业：计算机科学与技术**  **学生姓名：**  **学　　号：**  **指导教师：陈昆** |
|  |
| **日　　期： 2023年 6月 3日** |

**电 子 科 技 大 学**

**实 验 报 告**

**实验一**

1. **实验名称：词法分析器的设计与实现**
2. **实验学时：4**
3. **实验内容和目的：**

实现求n！的极小语言的词法分析程序

1. 二元式文件\*.dyd

(1) 二元式形式：

单词符号（长度为16） 种别（长度为2）（本实验中将产生错误的符号种别定为了99）

(2)每行后加：“...EOLN24”

(3)文件结尾加：“...EOF25”

2. 错误信息文件\*.err

(1)错误信息格式：

\*\*\*LINE：行号错误性质

（注：为了提升用户体验感以及界面美观感，结合实际编译器处理情况对报错格式作了调整）

具体格式为：

1. 非法字符：

LINE行号：illegal character ‘读到的字符’

1. 冒号不匹配

LINE行号：expected '=' after ':'

1. 标识符长度溢出

LINE行号：identifier '具体溢出标识符' is too long(Don't exceed 16 characters)

(2)词法需要报三种错误：非法字符、冒号不匹配、标识符长度溢出。

注意：进入每一阶段, 首先打开\*.err, 如果无错误, 则\*.err为空。

1. **实验原理：**

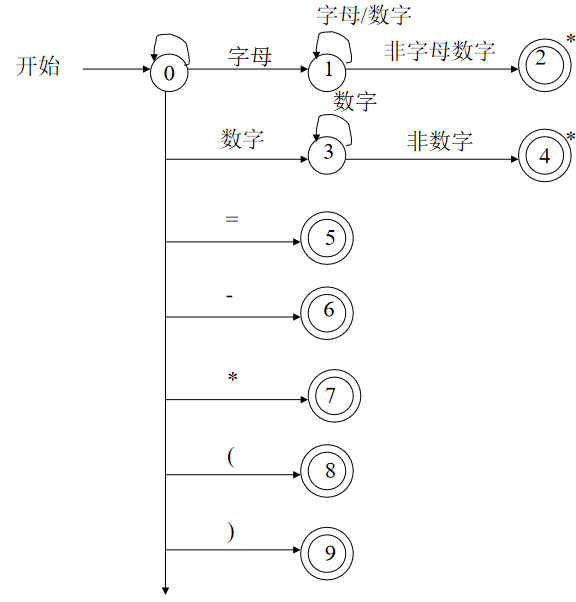
1. 编译程序要求对高级语言编写的源程序进行分析和合成，生成目标程序。词法分析是对源程序进行的首次分析，实现词法分析的程序为词法分析程序。

2. 词法分析的功能是从左到右逐个地扫描源程序字符串，按照词法规则识别出单词符号作为输出，对识别过程中发现的词法错误，输出相关信息。本实验单词符号与种别对照表如下所示：

表1 单词符号与种别对照表



3.状态转换图是有限有向图，是设计词法分析器的有效工具。本实验状态转换图如下所示：



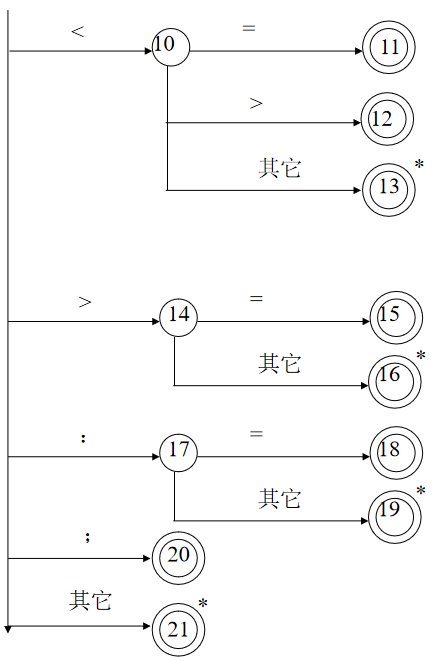


图1 状态转换图

1. **实验器材（设备、元器件）**

1. 操作系统：Windows 11

2. 开发工具：Visual Studio 2022

1. **实验步骤：**

算法流程图如下所示：

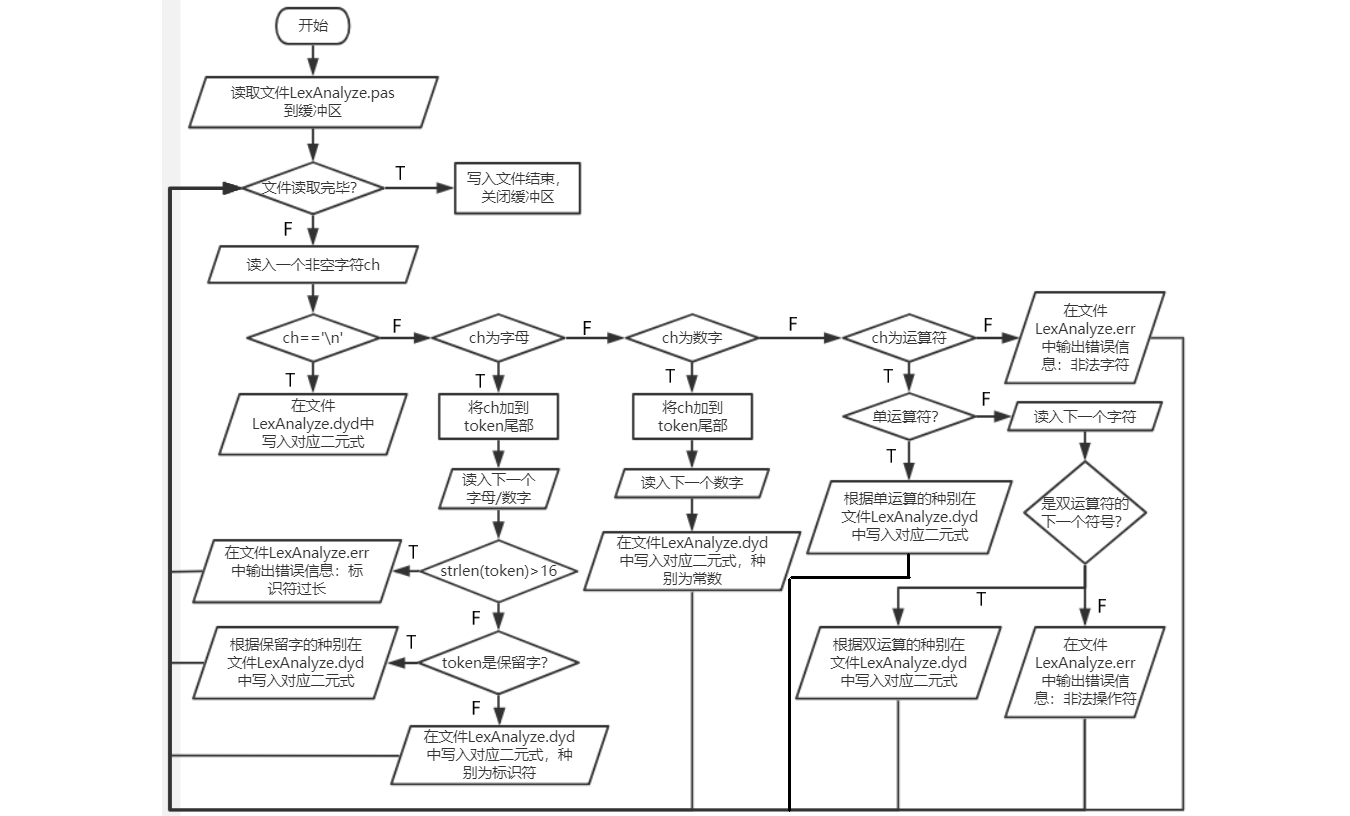


图2 算法流程图

1. **实验数据及结果分析：**
2. 正确的测试程序结果如下所示：

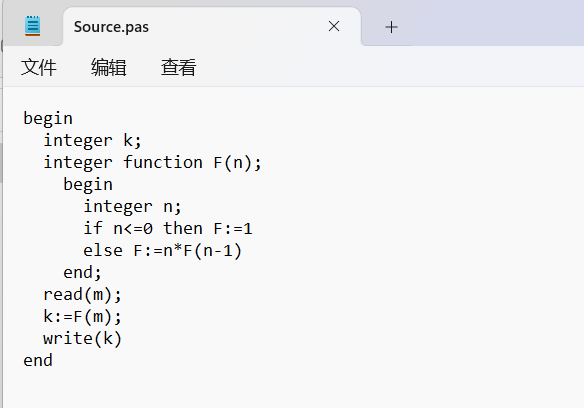


图3 测试程序

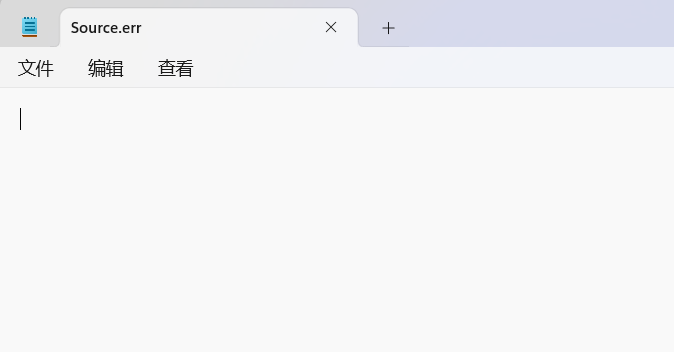


图4 错误信息文件

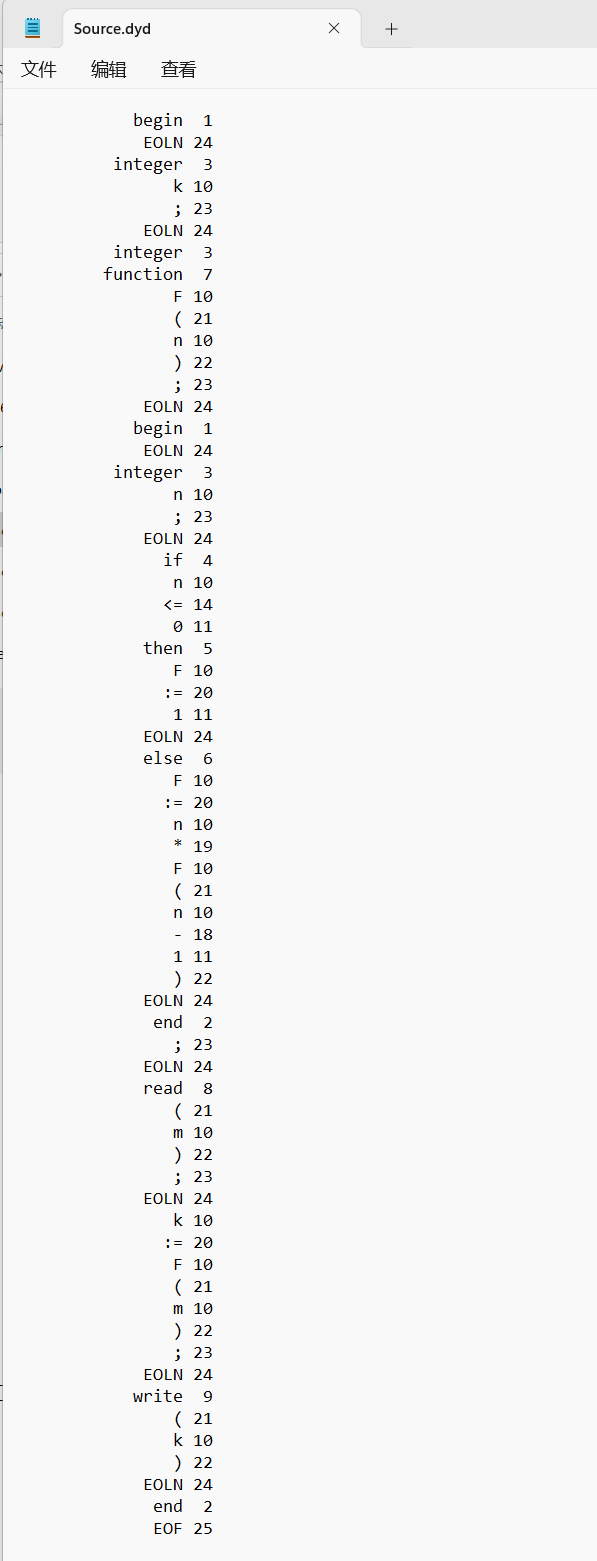


图5 二元式文件

1. 出错的测试程序结果如下所示：

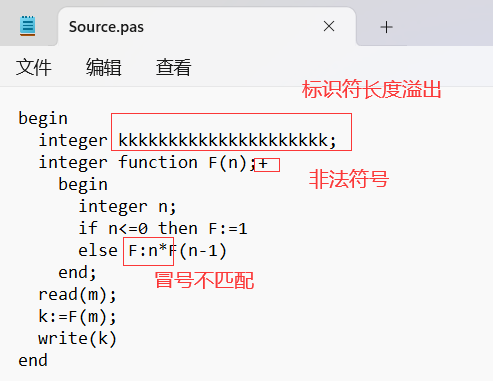


图6 测试文件

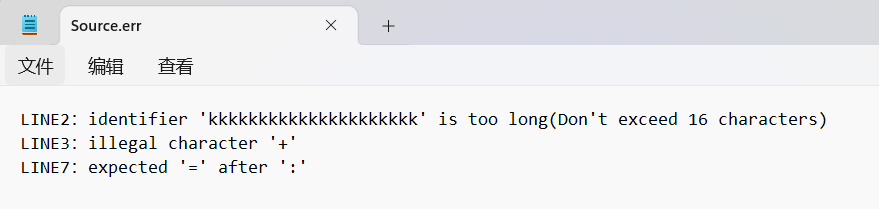


图7 错误信息文件

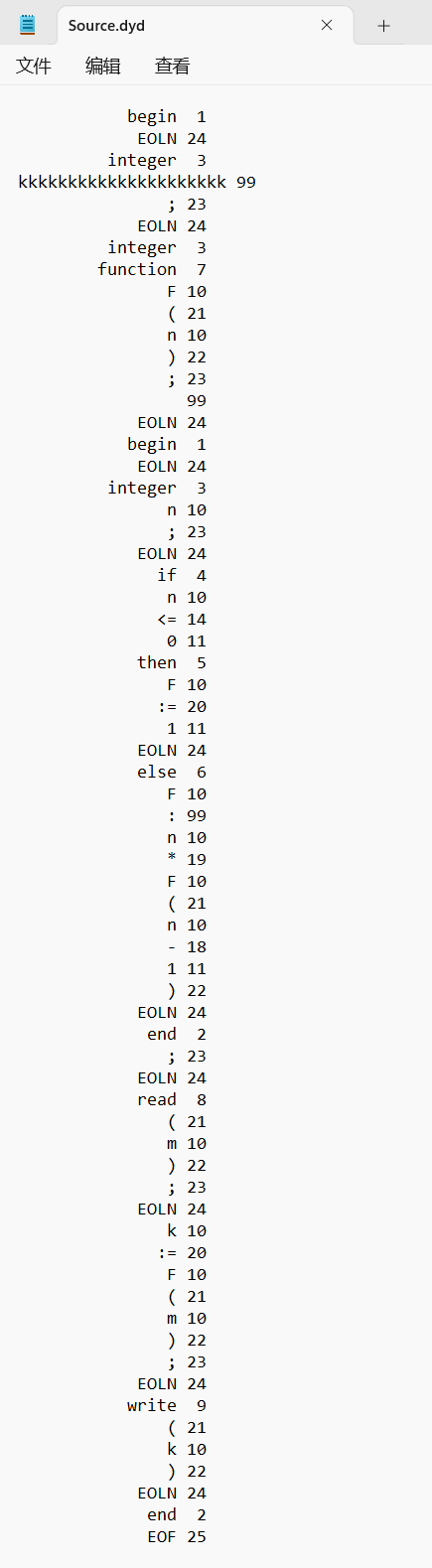


图8 二元式文件

1. **实验结论、心得体会和改进建议：**

（一）实验结论

1. 本实验程序可以对源程序进行词法分析。对正确的测试程序会生成二元式文件，错误信息文件为空；出错的测试程序错误信息文件会被写入错误信息和所在行数。

2. 本实验程序能正确地并一次性地报出所有错误。

（二）心得体会

1. 通过本次实验，精炼了调试技巧，并学习了很多c++字符串处理库函数，为下次语法分析实验积累了经验。

2. 经过实践，切身体会了从状态转换图到实际程序的实现方式，对于理解词法分析程序自动生成器的原理有极大帮助。

3. 当然也遇到了很多细节问题，如在最初使用getchar时回退不方便，因此后面改用了一次读进一行的方式，方便指针回退。也理解了课堂上说的回退是很麻烦的。

（三）改进建议

可以增加对于制表符\t的识别。

**电 子 科 技 大 学**

**实 验 报 告**

**实验二**

1. **实验名称：递归下降分析器的设计与实现**
2. **实验学时：4**
3. **实验内容和目的：**

根据给定的方法，编写相应的递归下降的语法分析程序，实现对词法分析后的单词序列的语法检查和程序结构的分析，生成相应的变量名表和过程名表，并将编译中语法检查出来的错误写入相应的文件。

语法错分类:

(1)缺少符号错;

(2)符号匹配错

(3)符号无定义或重复定义。

1. **实验原理：**

1. 语法分析是对源程序经过词法分析后转换成的单词流按方法规则进行判断，对能构成正确句子的单词流，给出相应的语法树；对不能构成正确句子的单词流判断其语法错误并做出相应处理。

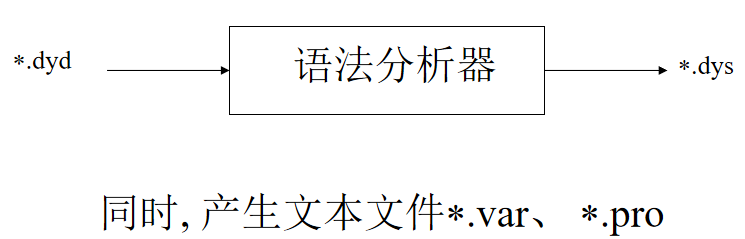


图 1

本实验的分析方法如下：

<程序>→<分程序>

<分程序>→begin <说明语句表>；<执行语句表> end

<说明语句表>→<说明语句>│<说明语句表> ；<说明语句>

<说明语句>→<变量说明>│<函数说明>

<变量说明>→integer <变量>

<变量>→<标识符>

<标识符>→<字母>│<标识符><字母>│<标识符><数字>

<字母>→a│b│c│d│e│f│g│h│i│j│k│l│m│n│o│p│q│r│s│ t│u│v│w│x│y│z

<数字>→0│1│2│3│4│5│6│7│8│9

<函数说明>→integer function <标识符>（<参数>）；<函数体>

<参数>→<变量>

<函数体>→begin <说明语句表>；<执行语句表> end

<执行语句表>→<执行语句>│<执行语句表>；<执行语句>

<执行语句>→<读语句>│<写语句>│<赋值语句>│<条件语句>

<读语句>→read(<变量>)

<写语句>→write(<变量>)

<赋值语句>→<变量>:=<算术表达式>

<算术表达式>→<算术表达式>-<项>│<项>

<项>→<项>\*<因子>│<因子>

<因子>→<变量>│<常数>│<函数调用>

<常数>→<无符号整数>

<无符号整数>→<数字>│<无符号整数><数字>

<条件语句>→if<条件表达式>then<执行语句>else <执行语句>

<条件表达式>→<算术表达式><关系运算符><算术表达式>

<关系运算符> →<│<=│>│>=│=│<>

消除左递归后的文法如下：

Program：程序 Program→SubProgram

SubProgram：分程序 SubProgram→begin StatementList;ExecutionList end

StatementList：说明语句表 StatementList→Statement StatementList'

StatementList'： StatementList'→;Statement StatementList'|ε

Statement：说明语句 Statement→integer statement’

Statement’： Statement’→function Identifier(Parameter);Function|Var

Var：变量 Var→Identifier

Identifier：标识符 Identifier→LetterIdentifier'

Identifier': Identifier'→Letter Identifier'|DigitalIdentifier'|ε

Letter：字母 Letter→a|...|h|i|...|z

Digit：数字 Digit→0|1|...|9

Parameter：参数 Parameter→Var

Function：函数体 Function→begin StatementList;ExectionList end

ExecutionList：执行语句表 ExecutionList→Execution ExecutionList'

ExecutionList’ ExecutionList'→;Execution ExecutionList'|ε

Execution：执行语句 Execution→Read|Write|Equal|If

Read：读语句 Read→read(Var)

Write：写语句 Write→write(Var)

Assign：赋值语句 Assign→Var:=ArithExpression

ArithExpression：算术表达式 ArithExpression→Item ArithExpression'

ArithExpression' ArithExpression'→-ItemArithExpression'|ε

Item：项 Item→Factor Item'

Item' Item'→\*Factor Item'|ε

Factor：因子 Factor→Var|Constant|FunctionCall

Constant：常数 Constant→Integer

Integer：无符号整数 Integer→Digit Integer'

Integer'： Integer'→Digit Integer'|ε

If：条件语句 If→if IfExpression then Execution else Execution

IfExpression：条件表达式

IfExpression→ArithExpression RelationalOperator ArithExpression

RelationalOperator：关系运算符 RelationalOperator→<|<=|>|>=|=|<>

FunctionCall：函数调用 FunctionCall→Identifier(ArithExpression)

2. 语法分析方法有自上而下和自下而上的分析方法。在不含左递归的文法G中，如果对每一个非终结符的所有候选式的第一个终结符都是两两不相交的（即无公共左因子），则可能构造出一个不带回溯的自上而下的分析程序，这个分析程序由一组递归过程组成，每个过程对应文法的一个非终结符。这样的分析程序称为递归下降分析程序。

消除左递归方法如下：

考虑如下含左递归的文法：

A→Aα1|Aα2|…|Aαm|β1|β2|…|βn

消除左递归后文法为：

A→(β1|β2|…|βn)A’ 用来产生以βi开头的句型/句子

A’→(α1|α2|…|αm)A’|ε

1. 变量名表

变量名vname: char(16)

所属过程vproc:char(16)

分类vkind: 0..1(0—变量、1—形参)

变量类型vtype: types

变量层次vlev: int

变量在变量表中的位置vadr: int(相对第一个变量而言)

types=(ints)

1. 过程名表

过程名pname: char(16)

过程类型ptype: types

过程层次plev: int

第一个变量在变量表中的位置fadr: int

最后一个变量在变量表中的位置ladr: int

5. 变量名表和过程名表的作用：连接声明与引用的桥梁，记住每个符号的相关信息，如作用域和绑定等，帮助编译的各个阶段正确有效地工作。

1. **实验器材（设备、元器件）**

1. 操作系统：Windows 11

2. 开发工具：Visual Studio 2022

1. **实验步骤：**

实验流程图如下所示：

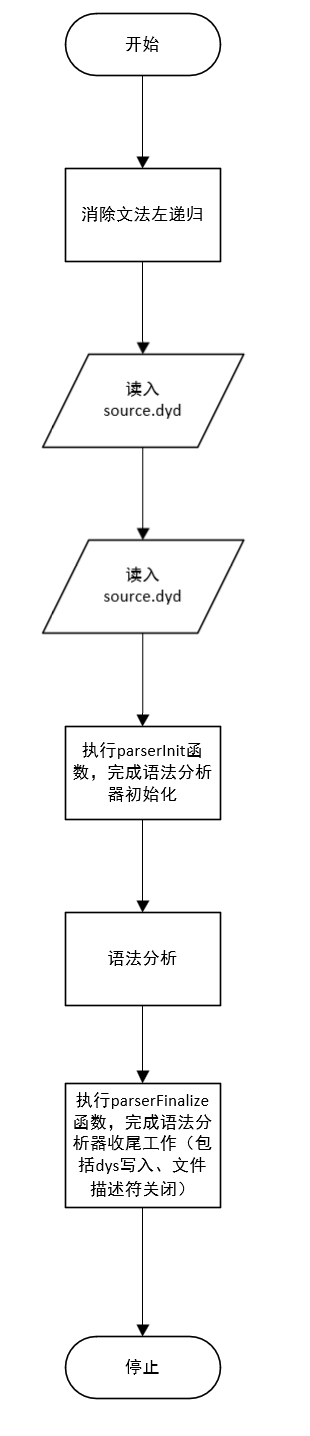


图2 实验流程图

1. **实验数据及结果分析：**
2. 直接使用测试程序

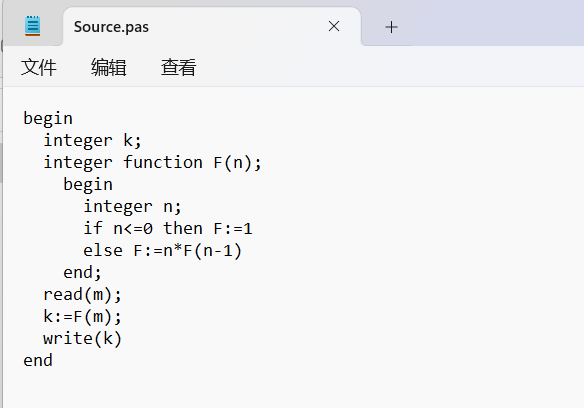


图3 测试程序

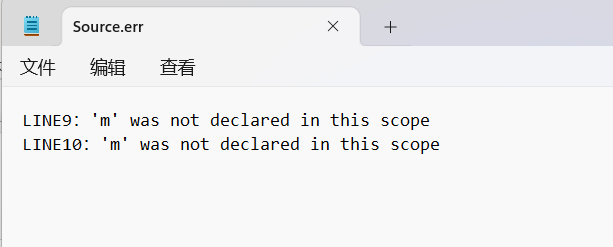


图4 错误信息文件

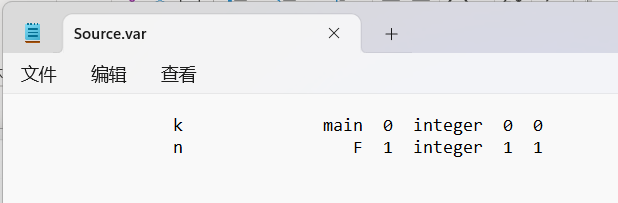


图5 变量表文件

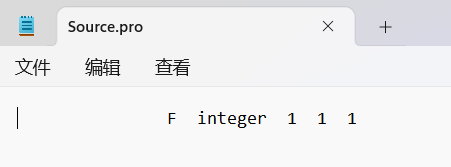


图6 过程表文件

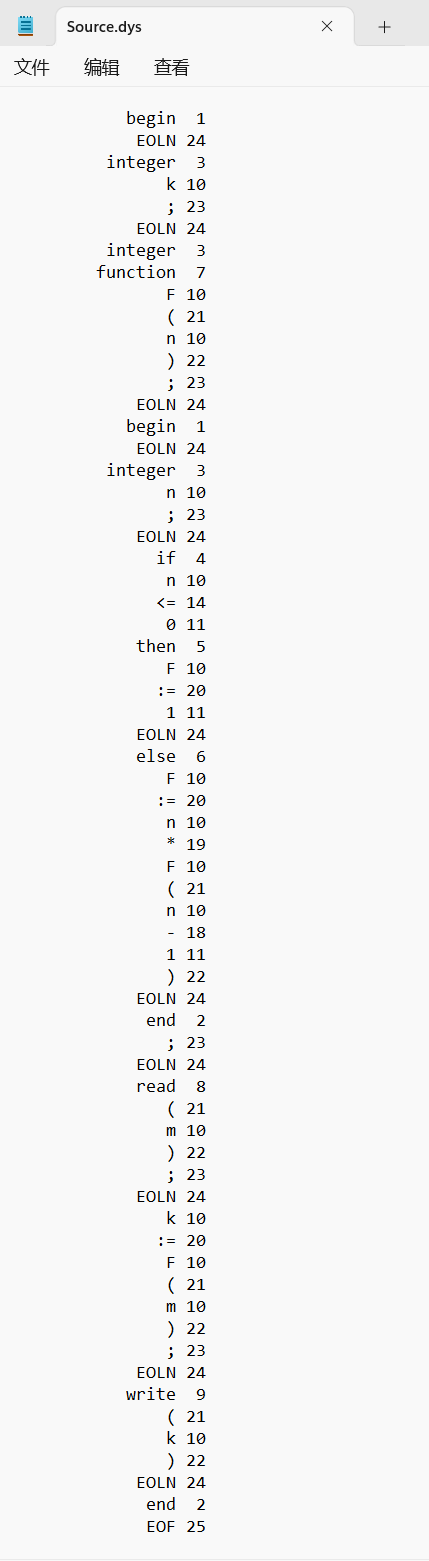


图7 二元式文件

结果分析：第九行和第十行均使用未声明的变量m，因此在语法分析器报出了相应错误。但由于未定义错误不属于重大语法错误，因此语法分析器并未停止分析。可以看到变量表中正常定义了变量k和形参n，过程表中正常定义了过程F。

1. 第二行改为：integer m；

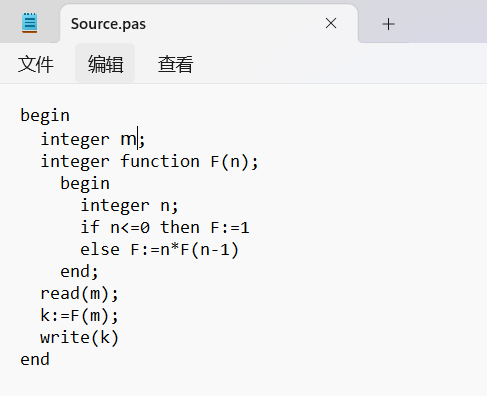


图8 测试程序

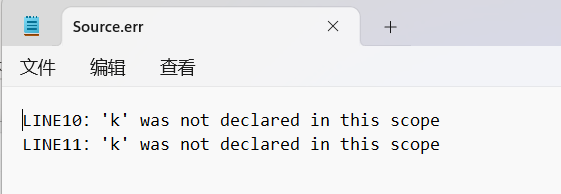


图9 错误信息文件

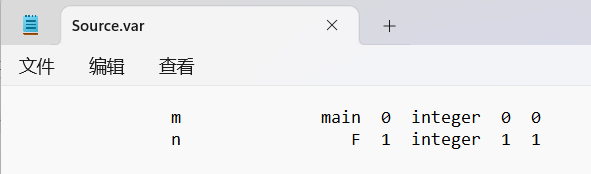


图10 变量表文件

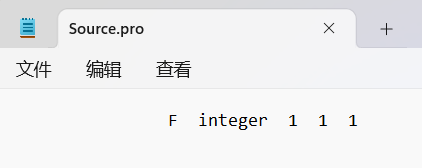


图11 过程表文件

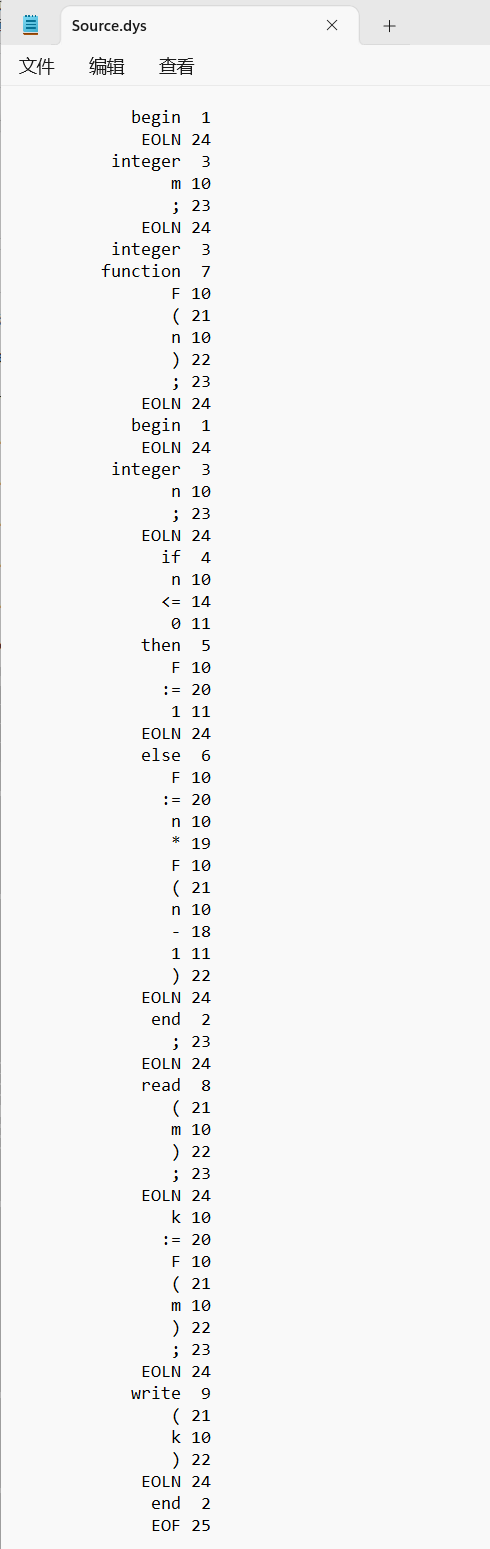


图12 二元式文件

结果分析：第十行和第十一行均使用未声明的变量k，因此在语法分析器报出了相应错误。但由于未定义错误不属于重大语法错误，因此语法分析器并未停止分析。可以看到变量表中正常定义了变量m和形参n，过程表中正常定义了过程F。

1. 第三行改为：integer funtion F(n)；

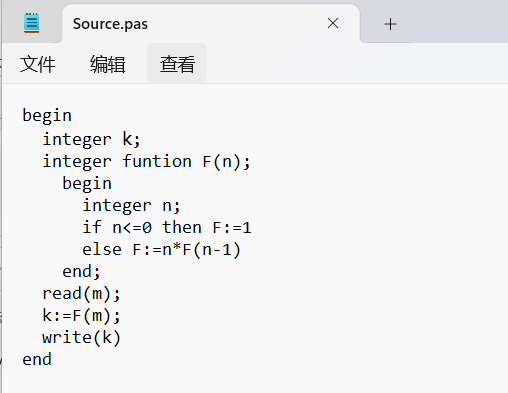


图13 测试程序

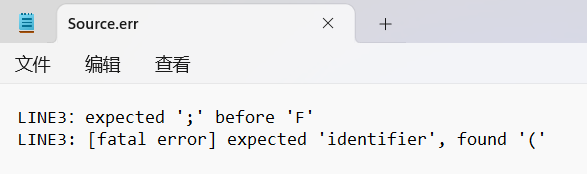


图13 错误信息文件

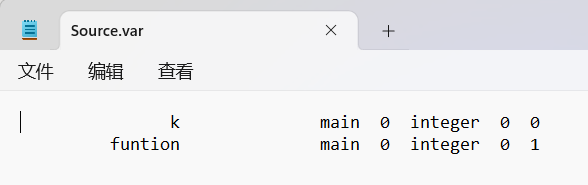


图14 变量表文件

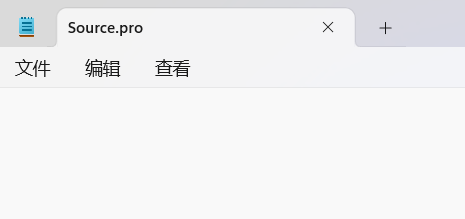


图15 过程表文件

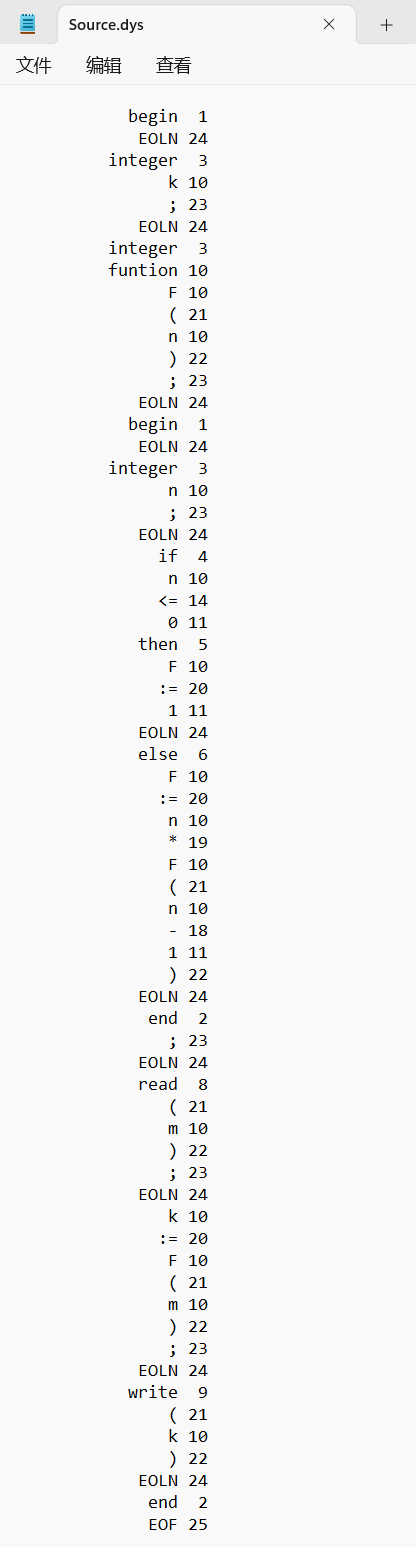


图16 二元式文件

结果分析：第三行在integer后语法分析器识别到了funtion，因为不在保留字里面所以被识别为变量写入变量表中。随后读入F，对于变量定义后面需要识别分号，但读到F，因为本行之前已经报过错，因此不再报错。随后读入（，因为不再是integer，所以语法分析器认为声明语句表已结束，便开始识别执行语句。而在执行语句的定义中，不满足read、write、if开头，在程序中处理为默认赋值语句，将识别标识符，但（又不是标识符，接下来的分析已没有意义，因此报出重大语法错误并结束语法分析。因为此时还没有声明过程，因此过程表为空。

1. 第九行改为：readm)

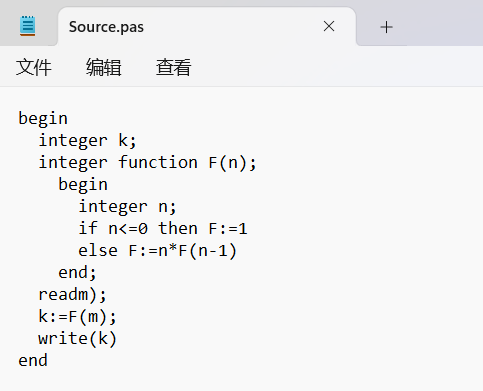


图17 测试程序

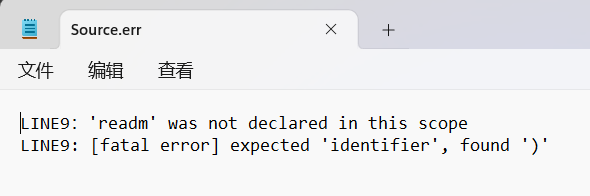


图18 错误信息文件

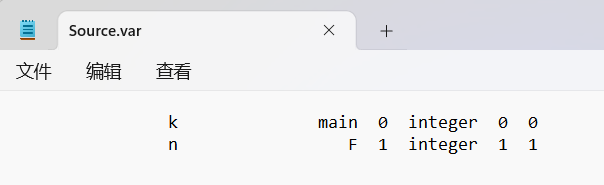


图19 变量表文件

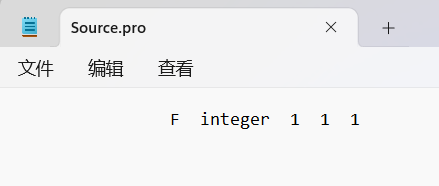


图20 过程表文件

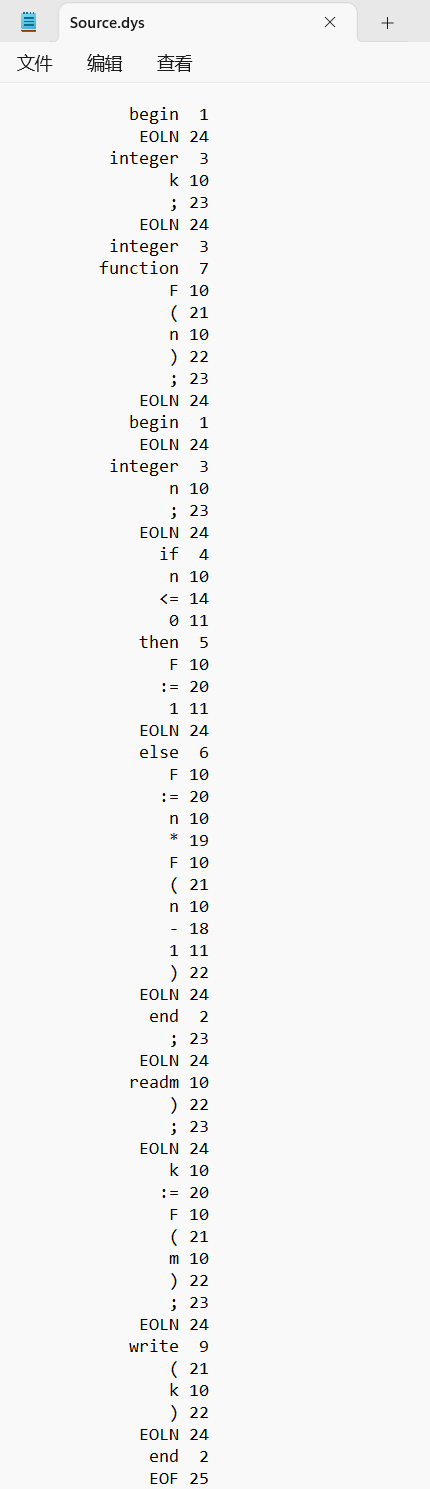


图21 二元式文件

结果分析：语法分析器在第九行读到了readm，因为此处已处于执行语句表，所以默认为执行语句中。在执行语句中，只有赋值语句以标识符开头，因此语法分析器认为这是一个赋值语句。接着读入），按照文法此处需要赋值符号，但因为本行之前已有报错，故不再报错。（因为错误处理程序默认是缺少赋值符号，因此下一个读入还是’）’）。按照文法此处需要标识符，但读到），继续语法分析已无意义，所以报出重大语法错误并退出。变量表和函数表的内容符合预期。

1. 第九行后加一行：integer m；

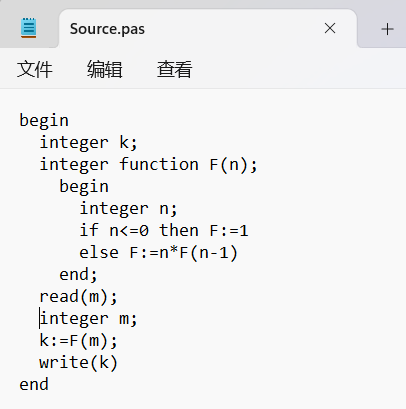


图22 测试程序

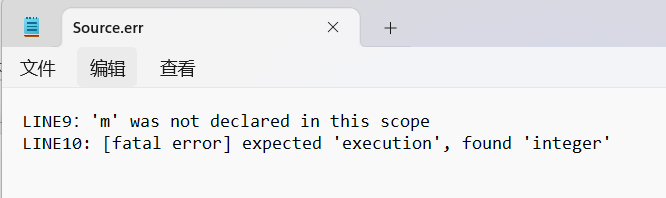


图23 错误信息文件

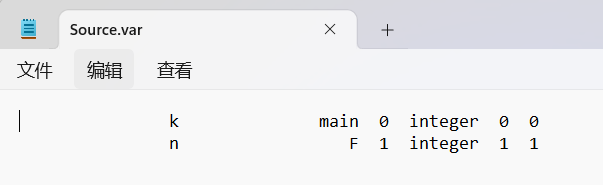


图24 变量表文件

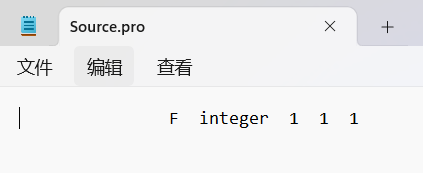


图25 过程表文件

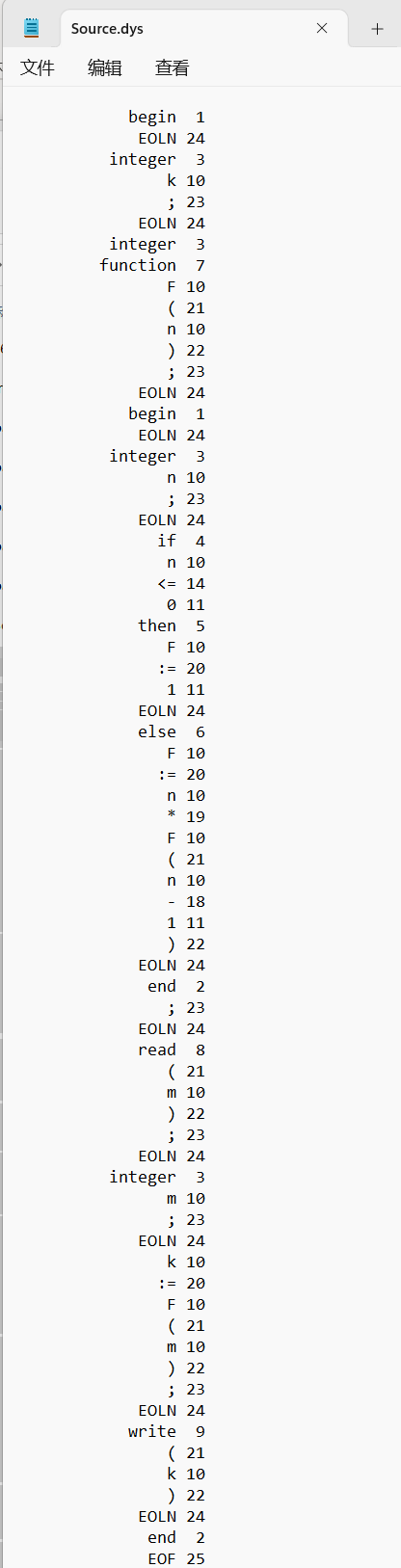


图26 二元式文件

结果分析：第九行语法分析器读到了未定义的变量m，正常报出未定义的错误。随后在第十行，因为当前仍处于执行语句表部分，语法分析器检查其不以read、write、if开头，便默认为赋值语句。期待读入标识符，但是读到integer，继续进行语法分析已无意义，因此报出重大语法错误并结束语法分析。变量表与过程表均符合预期。

1. **实验结论、心得体会和改进建议：**

（一）实验结论

1. 本实验程序可以对源程序进行语法分析，图中给出了出错行数及出错类型。

2. 在不出现重大语法错误时，一行只报一个错误。

（二）心得体会

1. 通过本次实验，精炼了调试技巧，并学习了很多c++字符串处理库函数，为以后开发积累了经验。

2. 经过实践，切身体会了从LL（1）文法到无回溯的递归下降语法分析程序的实现方式，对于理解语法分析程序自动生成器的原理有极大帮助。

3. 在错误处理时，有两个思路。其一是跳过当前符号继续分析（本程序采用），另一种是跳过该行继续分析。本来想采用后一种思路，但发现后一种思路便无法与重大语法错误分析对应（因为无论什么错误均可以跳过该行），因此本程序采用了第一种思路。

4. 在实际开发过程中，意识到实现合理的错误检测比实现递归下降分析本身困难得多。在设计过程中我也参考了gcc编译器的报错处理来对我的报错程序进行优化，使之更加人性化。

（三）改进建议

可以增加对于报错处理的讲解以及规定一些共识，不然在可能在理解错误种类方面花的额外精力会比较多。

附：词法分析源程序

parameters.h（定义各种常量，增加代码可读性）：

#pragma once

#define MAX\_FILENAME\_LEN 25

#define NONE 0

#define BEGIN 1

#define END 2

#define INTEGER 3

#define IF 4

#define THEN 5

#define ELSE 6

#define FUNCTION 7

#define READ 8

#define WRITE 9

#define ID 10

#define CONST 11

#define EQUAL 12

#define NEQUAL 13

#define LE 14

#define LT 15

#define GE 16

#define GT 17

#define SUB 18

#define MUL 19

#define ASSIGN 20

#define LPAR 21

#define RPAR 22

#define SEM 23

#define EOLN 24

#define EOF 25

#define ERROR 99

#define SIGN\_UNDEFINED\_ERROR 0

#define SIGN\_UNMATCHED\_ERROR 1

#define ID\_OVERFLOW\_ERROR 2

main.cpp：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<cstdio>

#include<cstring>

#include"parameters.h"

using namespace std;

FILE\* inFile = NULL;

FILE\* outFile = NULL;

FILE\* errFile = NULL;

const int MAX\_SIZE = 200;

const int TEXT\_MAX\_SIZE = 1000;

struct Tuple{

int c;

int value;

};

char ch, prevCh;

char strToken[MAX\_SIZE];

char text[TEXT\_MAX\_SIZE];

int idx, position;

int lineNum = 1;

void GetChar() {

ch = text[position++];

}

void GetBC() {

do GetChar(); while (ch == ' ');

}

void Concat() {

strToken[idx++] = ch;

}

bool IsLetter() {

if (ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z') {

return true;

}

else {

return false;

}

}

bool IsDigit() {

if (ch >= '0' && ch <= '9') {

return true;

}

else {

return false;

}

}

void Retract() {

position--;

}

int Reserve() {

if (strcmp(strToken, "begin") == 0) {

return BEGIN;

}

else if (strcmp(strToken, "end") == 0) {

return END;

}

else if (strcmp(strToken, "integer") == 0) {

return INTEGER;

}

else if (strcmp(strToken, "if") == 0) {

return IF;

}

else if (strcmp(strToken, "then") == 0) {

return THEN;

}

else if (strcmp(strToken, "else") == 0) {

return ELSE;

}

else if (strcmp(strToken, "function") == 0) {

return FUNCTION;

}

else if (strcmp(strToken, "read") == 0) {

return READ;

}

else if (strcmp(strToken, "write") == 0) {

return WRITE;

}

else {

return ID;

}

}

int InsertId() {

return 0;

}

int InsertConst() {

return 0;

}

int read() {

position = 0;

return fgets(text, 1000, inFile) == NULL ? 0 : 1;

}

void ProcError(int errorType) {

if (errorType == SIGN\_UNDEFINED\_ERROR) {

fprintf(errFile, "LINE%d：illegal character '%c'\n", lineNum, ch);

}

else if (errorType == SIGN\_UNMATCHED\_ERROR) {

fprintf(errFile, "LINE%d：expected '=' after ':'\n", lineNum);

}

else if (errorType == ID\_OVERFLOW\_ERROR) {

fprintf(errFile, "LINE%d：identifier '%s' is too long(Don't exceed 16 characters)\n", lineNum, strToken);

}

}

Tuple lex(FILE\* stream) {

memset(strToken, 0, sizeof(strToken));

idx = 0;

int code;

Tuple returnValue;

GetBC();

if(IsLetter()){

while (IsLetter() || IsDigit()) {

Concat();

GetChar();

}

Retract();

code = Reserve();

if (code == ID) {

if (strlen(strToken) > 16) {

ProcError(ID\_OVERFLOW\_ERROR);

returnValue.c = ERROR;

return returnValue;

}

else {

returnValue.c = ID;

returnValue.value = InsertId();

return returnValue;

}

}

else {

returnValue.c = code;

returnValue.value = 0;

return returnValue;

}

}

else if (IsDigit()) {

while (IsDigit()) {

Concat();

GetChar();

}

Retract();

returnValue.c = CONST;

returnValue.value = InsertConst();

return returnValue;

}

else if (ch == '=') {

returnValue.c = EQUAL;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == '<') {

Concat();

GetChar();

if (ch == '>') {

returnValue.c = NEQUAL;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == '=') {

returnValue.c = LE;

returnValue.value = 0;

Concat();

return returnValue;

}

Retract();

returnValue.c = LT;

returnValue.value = 0;

return returnValue;

}

else if (ch == '>') {

Concat();

GetChar();

if (ch == '=') {

returnValue.c = GE;

returnValue.value = 0;

Concat();

return returnValue;

}

Retract();

returnValue.c = GT;

returnValue.value = 0;

return returnValue;

}

else if (ch == '-') {

returnValue.c = SUB;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == '\*') {

returnValue.c = MUL;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == ':') {

Concat();

GetChar();

if (ch == '=') {

returnValue.c = ASSIGN;

returnValue.value = 0;

Concat();

return returnValue;

}

Retract();

ProcError(SIGN\_UNMATCHED\_ERROR);

returnValue.c = ERROR;

return returnValue;

}

else if (ch == '(') {

returnValue.c = LPAR;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == ')') {

returnValue.c = RPAR;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == ';') {

returnValue.c = SEM;

returnValue.value = 0;

Concat();

return returnValue;

}

else if (ch == '\n') {

strcpy(strToken, "EOLN");

returnValue.c = EOLN;

returnValue.value = 0;

lineNum++;

if(!read())returnValue.c = EOF;

return returnValue;

}

else if (ch == '\0') {

memset(strToken, 0, sizeof(strToken));

strcpy(strToken, "EOF");

returnValue.c = EOF;

return returnValue;

}

else {

ProcError(SIGN\_UNDEFINED\_ERROR);

returnValue.c = ERROR;

return returnValue;

}

}

void lexerInit();

void lexerFinalize();

int main() {

lexerInit();

Tuple ret;

ret.c = 0;

read();

while (ret.c != EOF) {

ret = lex(inFile);

fprintf(outFile, "%16s %2d\n", strToken, ret.c);

}

lexerFinalize();

return 0;

}

void lexerInit() {

char inFileName[MAX\_FILENAME\_LEN] = "Source.pas";

char outFileName[MAX\_FILENAME\_LEN] = "Source.dyd";

char errFileName[MAX\_FILENAME\_LEN] = "Source.err";

inFile = fopen(inFileName, "r");

outFile = fopen(outFileName, "w");

errFile = fopen(errFileName, "w");

}

void lexerFinalize() {

fclose(inFile);

fclose(outFile);

fclose(errFile);

}

附：语法分析源程序

parameters.h：（定义各种常量，增加代码可读性）

#pragma once

// Memory

#define MAX\_SIZE 100

#define MAX\_STACK\_SIZE 10

#define MAX\_LINE\_SIZE 25

#define MAX\_FILENAME\_LEN 25

// Kind

#define BEGIN 1

#define END 2

#define INTEGER 3

#define IF 4

#define THEN 5

#define ELSE 6

#define FUNCTION 7

#define READ 8

#define WRITE 9

#define ID 10

#define CONST 11

#define EQUAL 12

#define NEQUAL 13

#define LE 14

#define LT 15

#define GE 16

#define GT 17

#define SUB 18

#define MUL 19

#define ASSIGN 20

#define LPAR 21

#define RPAR 22

#define SEM 23

#define EOLN 24

#define EOF 25

// Var

#define VARIABLE 0

#define FORMAL\_PARAMETER 1

// ErrorProc

#define SIGN\_UNDEFINED\_ERROR 0

#define SIGN\_REDEFINED\_ERROR 1

#define SIGN\_UNMATCHED\_ERROR 2

#define SIGN\_ABSENCE\_ERROR 3

#define FATAL\_ERROR 4

main.cpp：

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<cstdio>

#include<cstring>

#include "parameters.h"

using namespace std;

int kind[MAX\_SIZE];

char input[MAX\_SIZE][17];

int pToken;

int varCount, proCount, inputCount;

int lineNum;

bool lineErr;

FILE\* inFile = NULL;

FILE\* outFile = NULL;

FILE\* errFile = NULL;

FILE\* varFile = NULL;

FILE\* proFile = NULL;

enum types {

integer

};

struct Var {

char name[17];

char proc[17];

int kind;

types type;

int level;

int addr;

}vars[MAX\_SIZE], pVar;

struct Proc {

char name[17];

types type;

int level;

int faddr;

int laddr;

int varNum;

char parameter[20];

bool parameterIsDefined;

}procs[MAX\_SIZE], pProc;

bool nextToken();

bool isVarExisted(char\* vname, char\* vporc, int vkind);

bool isProcExisted(char\* pname);

void ErrorProc(int errorType, const char\* token);

void Program();

void SubProgram();

void StatementList();

void \_StatementList();

void Statement();

void \_Statement();

void Parameter();

void Function();

void ExecutionList();

void \_ExecutionList();

void Execution();

void Read();

void Write();

void Assign();

void ArithExpression();

void \_ArithExpression();

void Item();

void \_Item();

void Factor();

void Constant();

void If();

void IfExpression();

void RelationalOperator();

void FunctionCall();

void Var();

void Identifier();

void parserInit();

void parserFinalize();

int main() {

parserInit();

Program();

parserFinalize();

return 0;

}

bool nextToken() {

pToken++;

if (kind[pToken] == EOF) {

return false;

}

while (kind[pToken] == EOLN) {

pToken++;

lineNum++;

lineErr = false;

}

return true;

}

bool isVarExisted(char\* vname, char\* vproc, int vkind) {

for (int i = 0; i < varCount; i++) {

if (strcmp(vname, vars[i].name) == 0 && strcmp(vproc, vars[i].proc) == 0 && vkind == vars[i].kind) {

return true;

}

}

return false;

}

bool isProcExisted(char\* pname) {

for (int i = 0; i < proCount; i++) {

if (strcmp(pname, procs[i].name) == 0) {

return true;

}

}

return false;

}

void ErrorProc(int errorType, const char\* token) {

if (errorType == SIGN\_UNDEFINED\_ERROR && !lineErr) {

fprintf(errFile, "LINE%d：'%s' was not declared in this scope\n", lineNum, token);

}

else if (errorType == SIGN\_REDEFINED\_ERROR && !lineErr) {

fprintf(errFile, "LINE%d: redeclaration of '%s'\n", lineNum, input[pToken]);

}

else if (errorType == SIGN\_UNMATCHED\_ERROR && !lineErr) {

fprintf(errFile, "LINE%d：expected '%s', found '%s'\n", lineNum, token, input[pToken]);

}

else if (errorType == SIGN\_ABSENCE\_ERROR && !lineErr) {

fprintf(errFile, "LINE%d：expected '%s' before '%s'\n", lineNum, token, input[pToken]);

}

else if (errorType == FATAL\_ERROR) {

fprintf(errFile, "LINE%d: [fatal error] expected '%s', found '%s'\n", lineNum, token, input[pToken]);

parserFinalize();

exit(0);

}

lineErr = true;

}

void Program() {

SubProgram();

}

void SubProgram() {

if (kind[pToken] == BEGIN) {

nextToken();

}

else {

if (kind[pToken] == INTEGER) {

ErrorProc(SIGN\_ABSENCE\_ERROR, "begin");

}

else {

ErrorProc(SIGN\_UNMATCHED\_ERROR, "begin");

nextToken();

}

}

StatementList();

ExecutionList();

if (kind[pToken] == END) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "end");

}

}

void StatementList() {

Statement();

if (kind[pToken] == SEM) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ";");

lineErr = false;

}

\_StatementList();

}

void \_StatementList() {

if (kind[pToken] == INTEGER) {

Statement();

if (kind[pToken] == SEM) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ";");

}

\_StatementList();

}

}

void Statement() {

if (kind[pToken] == INTEGER) {

nextToken();

}

else {

ErrorProc(SIGN\_UNMATCHED\_ERROR, "integer");

nextToken();

}

\_Statement();

}

void \_Statement() {

if (kind[pToken] == FUNCTION) {

nextToken();

Proc backupProc = pProc;

strcpy(pProc.name, input[pToken]);

pProc.type = integer;

pProc.level++;

pProc.varNum = 0;

pProc.parameterIsDefined = false;

if (isProcExisted(input[pToken])) {

ErrorProc(SIGN\_REDEFINED\_ERROR, "");

}

Identifier();

if (kind[pToken] == LPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "(");

}

strcpy(pProc.parameter, input[pToken]);

Parameter();

if (kind[pToken] == RPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ")");

}

if (kind[pToken] == SEM) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ";");

}

Function();

pProc = backupProc;

}

else {

strcpy(pVar.name, input[pToken]);

strcpy(pVar.proc, pProc.name);

if (strcmp(input[pToken], pProc.parameter) == 0) {

pVar.kind = FORMAL\_PARAMETER;

pProc.parameterIsDefined = true;

}

else {

pVar.kind = VARIABLE;

}

pVar.type = integer;

pVar.level = pProc.level;

pVar.addr = varCount;

if (isVarExisted(pVar.name, pVar.proc, pVar.kind)) {

ErrorProc(SIGN\_REDEFINED\_ERROR, "");

Var();

}

else {

if (pProc.varNum == 0) {

pProc.faddr = pVar.addr;

}

pProc.laddr = pVar.addr;

pProc.varNum++;

Var();

vars[varCount++] = pVar;

}

}

}

void Parameter() {

Var();

}

void Function() {

if (kind[pToken] == BEGIN) {

nextToken();

}

else {

if (kind[pToken] == INTEGER) {

ErrorProc(SIGN\_ABSENCE\_ERROR, "begin");

}

else {

ErrorProc(SIGN\_UNMATCHED\_ERROR, "begin");

nextToken();

}

}

StatementList();

if (!pProc.parameterIsDefined) {

ErrorProc(SIGN\_UNDEFINED\_ERROR, pProc.parameter);

}

procs[proCount++] = pProc;

ExecutionList();

if (kind[pToken] == END) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "end");

}

}

void ExecutionList() {

Execution();

\_ExecutionList();

}

void \_ExecutionList() {

if (kind[pToken] == SEM) {

nextToken();

Execution();

\_ExecutionList();

}

else if (kind[pToken] != END) {

ErrorProc(SIGN\_ABSENCE\_ERROR, ";");

Execution();

\_ExecutionList();

}

}

void Execution() {

if (kind[pToken] == READ) {

nextToken();

Read();

}

else if (kind[pToken] == WRITE) {

nextToken();

Write();

}

else if (kind[pToken] == IF) {

nextToken();

If();

}

else if (kind[pToken] == ID) {

Assign();

}

else {

ErrorProc(FATAL\_ERROR, "execution");

}

}

void Read() {

if (kind[pToken] == LPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "(");

}

Var();

if (!isVarExisted(input[pToken - 1], pProc.name, FORMAL\_PARAMETER) && !isVarExisted(input[pToken - 1], pProc.name, VARIABLE)) {

pToken--;

ErrorProc(SIGN\_UNDEFINED\_ERROR, input[pToken]);

pToken++;

}

if (kind[pToken] == RPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ")");

}

}

void Write() {

if (kind[pToken] == LPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "(");

}

Var();

if (!isVarExisted(input[pToken - 1], pProc.name, FORMAL\_PARAMETER) && !isVarExisted(input[pToken - 1], pProc.name, VARIABLE)) {

pToken--;

ErrorProc(SIGN\_UNDEFINED\_ERROR, input[pToken]);

pToken++;

}

if (kind[pToken] == RPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ")");

}

}

void Assign() {

Var();

if (!isVarExisted(input[pToken - 1], pProc.name, FORMAL\_PARAMETER) && !isVarExisted(input[pToken - 1], pProc.name, VARIABLE) && !isProcExisted(input[pToken - 1])) {

pToken--;

ErrorProc(SIGN\_UNDEFINED\_ERROR, input[pToken]);

pToken++;

}

if (kind[pToken] == ASSIGN) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ":=");

}

ArithExpression();

}

void ArithExpression() {

Item();

\_ArithExpression();

}

void \_ArithExpression() {

if (kind[pToken] == SUB) {

nextToken();

Item();

\_ArithExpression();

}

}

void Item() {

Factor();

\_Item();

}

void \_Item() {

if (kind[pToken] == MUL) {

nextToken();

Factor();

\_Item();

}

}

void Factor() {

if (kind[pToken] == CONST) {

Constant();

}

else if (isProcExisted(input[pToken])) {

FunctionCall();

}

else {

Var();

if (!isVarExisted(input[pToken - 1], pProc.name, FORMAL\_PARAMETER) && !isVarExisted(input[pToken - 1], pProc.name, VARIABLE)) {

pToken--;

ErrorProc(SIGN\_UNDEFINED\_ERROR, input[pToken]);

pToken++;

}

}

}

void Constant() {

if (kind[pToken] == CONST) {

nextToken();

}

else {

ErrorProc(FATAL\_ERROR, "constant");

}

}

void If() {

IfExpression();

if (kind[pToken] == THEN) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "then");

}

Execution();

if (kind[pToken] == ELSE) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "else");

}

Execution();

}

void IfExpression() {

ArithExpression();

RelationalOperator();

ArithExpression();

}

void RelationalOperator() {

if (kind[pToken] == EQUAL || kind[pToken] == NEQUAL || kind[pToken] == LE || kind[pToken] == LT || kind[pToken] == GE || kind[pToken] == GT) {

nextToken();

}

else {

ErrorProc(FATAL\_ERROR, "relational operator");

}

}

void FunctionCall() {

Identifier();

if (!isProcExisted(input[pToken - 1])) {

pToken--;

ErrorProc(SIGN\_UNDEFINED\_ERROR, input[pToken]);

pToken++;

}

if (kind[pToken] == LPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, "(");

}

ArithExpression();

if (kind[pToken] == RPAR) {

nextToken();

}

else {

ErrorProc(SIGN\_ABSENCE\_ERROR, ")");

}

}

void Var() {

Identifier();

}

void Identifier() {

if (kind[pToken] == ID) {

nextToken();

}

else {

ErrorProc(FATAL\_ERROR, "identifier");

}

}

void parserInit() {

char inFileName[MAX\_FILENAME\_LEN] = "Source.dyd";

char outFileName[MAX\_FILENAME\_LEN] = "Source.dys";

char errFileName[MAX\_FILENAME\_LEN] = "Source.err";

char varFileName[MAX\_FILENAME\_LEN] = "Source.var";

char proFileName[MAX\_FILENAME\_LEN] = "Source.pro";

inFile = fopen(inFileName, "r");

outFile = fopen(outFileName, "w");

errFile = fopen(errFileName, "w");

varFile = fopen(varFileName, "w");

proFile = fopen(proFileName, "w");

pToken = 0;

lineNum = 1;

strcpy(pProc.name, "main");

pProc.level = 0;

inputCount = 0;

varCount = 0;

proCount = 0;

while (!feof(inFile)) {

int i;

char lineStr[MAX\_LINE\_SIZE];

char lineName[MAX\_LINE\_SIZE], lineType[MAX\_LINE\_SIZE];

memset(lineStr, 0, sizeof(lineStr));

memset(lineName, 0, sizeof(lineName));

memset(lineType, 0, sizeof(lineType));

fgets(lineStr, MAX\_LINE\_SIZE, inFile);

strncpy(lineName, lineStr, 16);

strncpy(lineType, lineStr + 17, 2);

lineType[2] = '\0';

for (i = 0; lineName[i] == ' '; i++);

strcpy(input[inputCount], lineName + i);

kind[inputCount++] = atoi(lineType);

}

inputCount--;

}

void parserFinalize() {

for (int i = 0; i < varCount; i++) {

fprintf(varFile, "%16s %16s %d %s %d %d\n", vars[i].name, vars[i].proc, vars[i].kind, "integer", vars[i].level, vars[i].addr);

}

for (int i = 0; i < proCount; i++) {

fprintf(proFile, "%16s %s %d %d %d\n", procs[i].name, "integer", procs[i].level, procs[i].faddr, procs[i].laddr);

}

fseek(inFile, 0, 0);

int c = fgetc(inFile);

while (c != -1) {

fputc(c, outFile);

c = fgetc(inFile);

}

fclose(inFile);

fclose(outFile);

fclose(errFile);

fclose(varFile);

fclose(proFile);

}