

**课 程 实 验 报 告**

**课程名称： 编 译 原 理 实 验**

**专业班级： 计科校交1803**

**学 号： U201810824**

**姓 名： 孟彦康**

**指导教师： 万瑶**

**报告日期： 2021年 6月 20 日**

**计算机科学与技术学院**

**目录**

**[1 实验一 DavidC语言定义 2](#_Toc17018)**

[1.1 实验内容 2](#_Toc15053)

[1.2 实验类型 2](#_Toc12367)

[1.3 实验目标与要求 3](#_Toc32009)

[1.4 语言设计 3](#_Toc29963)

**[2 实验二 词法分析器设计与实现 6](#_Toc171)**

[2.1 单词文法描述 6](#_Toc6214)

[2.2 词法分析器设计 7](#_Toc8623)

[2.3 词法分析器实现结果 10](#_Toc21938)

**[3 实验三 自定义语言语法分析器设计与实现 11](#_Toc17953)**

[3.1 语法文法描述 11](#_Toc6109)

[3.2 语法分析器设计 17](#_Toc8253)

[3.3 语法分析器实现结果 25](#_Toc6742)

**[4 实验四 符号表管理和属性计算 26](#_Toc3727)**

[4.1 符号表结构 26](#_Toc30153)

[4.2 符号表显示 27](#_Toc29215)

[4.3 属性计算 28](#_Toc11774)

[4.4 符号表管理和属性计算结果 30](#_Toc21228)

**[5 实验五 静态语义分析 30](#_Toc22764)**

[5.1 错误类型定义 30](#_Toc11585)

[5.2 语义分析实现 30](#_Toc28471)

[5.3 静态语义分析结果 41](#_Toc22181)

**[6 实验六 中间代码生成 41](#_Toc6615)**

[6.1 中间代码生成格式定义 41](#_Toc8065)

[6.2 中间代码生成翻译模式 42](#_Toc11703)

[6.2 中间代码生成过程 43](#_Toc16040)

[6.3 中间代码生成结果 51](#_Toc19853)

**[7 总结 51](#_Toc15972)**

[7.1 实验完成情况 51](#_Toc11071)

[7.2 实验感想 52](#_Toc7867)

[7.3 展望 52](#_Toc28041)

# 1 实验一 DavidC语言定义

## 实验内容

自定义一个高级程序设计语言的词法和语法规则。

1. 可以基于某种熟悉的高级程序语言，设计一个自定义的高级语言子语言，完成其词法和语法规则。
2. 也可以选用Decaf语言作为源语言。

## 实验类型

设计型实验。

## 实验目标与要求

1. 为语言命名；
2. 标识符、常数、字符串等单词的文法；
3. 符号集、保留字集、运算符、界符；
4. 说明语句文法；
5. 赋值语句文法（简单赋值）；
6. 表达式求值文法（简单算术运算，包括++， --）；
7. 分支语句文法；
8. 循环语句文法；
9. 输入语句、输出语句文法；
10. 过程或函数调用语句文法；
11. 注释（行、块）。

## 语言设计

1. 文法所用到的元符号表：

表 1-1元符号信息表

|  |  |
| --- | --- |
| **x**（粗体） | 表示 x 是一个终结符（即“单词”）。 除个别关键字以外，本节中的终结符名字均为小写字母。 |
| y(常规) | 表示y是一个非终结符。非终结符的名字均为首字母大写。 |
| <x> | 表示0或1个x的出现，也就是说，x是可选的。 |
| x\* | 表示0、1或多个x的出现。 |
| x+ | 表示一个或多个似逗号分隔的x。 |
| | | 表示并列关系。 |
| ε | 表示没有，即不存在任何符号。 |

1. 标识符文法G[identifier]：

**identifier** ::=Fwords Mwords+ - **keyword**

fwords ::=[**a-z**] | [**A-Z**] | **\_**

mwords ::=fwords | [**0-9**]

1. 常量文法

1、整型常量G[intConstant]：

**intConstant** ::=Num+ | Fxnum Hnum+

Num ::=[**0-9**]

Fxnum ::=**0x** | **0X**

Hnum ::=Num | [**a-fA-F**]

2、布尔常量G[boolConstant]：

**boolConstant**::=**ture** | **false** | **0** | **1**

3、字符串常量G[stringConstant]：

**stingConstant**::=**” str****”**

**str** ::=Word+

**enter** ::=**换行符**

Escape ::=**\t** | **\n** | **\”**

Word ::=[**a-zA-Z0-9**]|[[`](mailto:[`~!@#$%^&*()_+-=,.<>/?/|:;])**[~!@#$%^&\*()\_+-=,.<>/?|:;](mailto:[`~!@#$%^&*()_+-=,.<>/?/|:;])**]|**空格**|**制表符** |Escape

1. 符号集、运算符、界符

**+ - \* / % < <= > >= = == != && || ! ; , . [ ] ( ) { }**

1. 保留字集G[keyword]：

**keyword** ::= **bool** | **break** | **if** | **else** | **for** | **while** | **NULL** | **void** | **char** | **int** | **return** | **printf** | **scanf** | **getchar** | **gets**

表 1-2保留字集表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| bool | break | if | else | for |
| while | NULL | void | char | int |
| return | printf | scanf | getchar | gets |

1. 参考Decaf/Mind语言，进行一个简单C语言文法设计。

**语言名称：**David-C

**文法设计：**

Program ::=FunctionDef\* MainDef

MainDef ::=Type main(<int argc, char \*argv[] | void>) StmBlock

FunctionDef ::=Type identifier ( Formals ) StmtBlock

Formals ::=Variable (, Variable)\* | ε

Variable ::=Type identifier | Type identifier[intConstant]

VariableDef ::=Variable ;

Type ::=int | bool | char | void | Type\*

StmtBlock ::={ Stmt\*}

Annotation ::=// str enter

AnnBlock ::=/\* Strword \*/

Strwords ::=(str | enter)\* - \*/

Stmt ::=VariableDef | SimpleStmt ; | IfStmt | WhileStmt | ForStmt | BreakStmt ; | PrintStmt ;| StmtBlock | Annotation | AnnBlock

BCStmt ::=Stmt | BreakStmt ; |ContinueStmt ; | BCIf

SimpleStmt ::=LValue = Expr | ε

LValue ::= identifier | Expr [ Expr ]

Call ::=identifier ( Actuals )

Actuals ::=Expr (, Expr)\* | ε

ForStmt ::=for( SimpleStmt;BoolExpr;SimpleStmt )BCStmt

WhileStmt ::=while ( BoolExpr ) Stmt

IfStmt ::=if ( BoolExpr ) Stmt <else Stmt>

BCIf ::=if ( BoolExpr ) BCStmt <else BCStmt>

ReturnStmt ::=return | return Expr

BreakStmt ::=break

ContinueStmt ::=continue

PrintStmt ::=printf (Expr (, Expr)\*)

ScanfStmt ::=scanf (<stringConstant ,> Scanf (, Scanf)\*) | getchar( identifier ) | gets( identifier )

Scanf ::=& LValue | LValue

BoolExpr ::=Expr

Expr ::=Constant | LValue | Call | ( Expr ) | Expr + Expr | Expr - Expr | Expr \* Expr | Expr / Expr | Expr % Expr | - Expr | Expr < Expr | Expr <= Expr | Expr > Expr | Expr >= Expr | Expr == Expr | Expr != Expr | Expr && Expr | Expr || Expr | ! Expr | Expr ? Expr : Expr| ( Type ) Expr | Expr++ | ++Expr | Expr-- | --Expr

Constant ::=intConstant | boolConstant | stringConstant | **NULL**

表 1-3文法设计相关说明

|  |  |
| --- | --- |
| 名称 | 说明 |
| Program | 程序 |
| MainDef | 主函数定义 |
| FunctionDef | 函数定义 |
| Formals | 函数定义中的参数 |
| Variable | 变量声明 |
| VariableDef | 变量定义语句 |
| Type | 类型 |
| StmtBlock | 语句块 |
| Stmt | 语句 |
| SimpleStmt | 简单赋值语句 |
| LValue | 可赋值变量 |
| Call | 函数调用 |
| Actuals | 调用函数时传入的参数 |
| ForStmt | for循环语句 |
| WhileStmt | while循环语句 |
| IfStmt | if条件判断语句 |
| ReturnStmt | 函数调用返回语句 |
| BreakStmt | 跳出循环语句 |
| PrintStmt | 输出语句 |
| ScanfStmt | 输入语句 |
| Scanf | 需要接收输入内容的变量 |
| BoolExpr | 真值表达式 |
| Expr | 表达式 |
| Constant | 常量 |
| BCStmt | 含有break和continue语句的操作 |
| ContinueStmt | 继续循环语句 |
| BCIf | 含有break和continue语句的判断语句 |

# 2 实验二 词法分析器设计与实现

## 2.1 单词文法描述

按照语法定义列出所有的终结符以及非终结符，如表 2-1 词法符号表所示：

表 2-1 词法符号表

|  |  |  |
| --- | --- | --- |
| 单词符号类型 | 单词种类码 | 正则表达式 |
| {INT} | INT | {[0-9]}+|{’’0X’’|”0x”}{{0-9}|[A-Fa-f]} |
| {FLOAT} | FLOAT | [0-9]\*[.][0-9]+([eE][-+]?([1-9][0-9\*]|[0]))? |
| {CHAR} | CHAR | ['][^\n]['] |
| {STRING} | STRING | ["][^\n]\*["] |
| {BOOL} | BOOL | ("true"|"false") |
| {type} | TYPE | ("bool"|"void"|"char"|"int"|"string"|"float") |
| “return” | RETURN |  |
| “if” | IF |  |
| “else” | ELSE |  |
| “while” | WHILE |  |
| “break” | BREAK |  |
| “continue” | CONTINUE |  |
| “for” | FOR |  |
| “switch” | SWITCH |  |
| “case” | CASE |  |
| “struct” | STRUCT |  |
| “default” | DEFAULT |  |
| {ID} | ID | {[a-zA-Z]|”\_”}{{a-zA-Z|”\_”}|[0-9]} |
| {ERRORID} | ERRORID | [0-9][A-Za-z0-9]\* |
| “:” | COLON |  |
| “;” | SEMI |  |
| “,” | COMMA |  |
| {compare} | RELOP | ("<"|">"|"<="|">="|"=="|"!=") |
| “=” | ASSIGNOP |  |
| “+” | PLUS |  |
| “-” | MINUS |  |
| “\*” | STAR |  |
| “/” | DIV |  |
| “.” | DOT |  |
| “%” | MOD |  |
| “&&” | AND |  |
| “||” | OR |  |
| “!” | NOT |  |
| “++” | AUTOPLUS |  |
| “--” | AUTOMINUS |  |
| “+=” | PLUSASSIGNOP |  |
| “-=” | MINUSASSIGNOP |  |
| “\*=” | STARASSIGNOP |  |
| “/=” | DIVASSIGNOP |  |
| “%=” | MODASSIGNOP |  |
| “(” | LP |  |
| “)” | RP |  |
| “{” | LC |  |
| “}” | RC |  |
| “[” | LB |  |
| “]” | RB |  |
|  | 单行注释 | \/\/[^\n]\* |
|  | 注释块开始符 | “/\*” |
|  | 注释块结束符 | <COMMENT>"\*/" |
|  | 注释内容 | <COMMENT>([^\*]|\n)+|. |

## 2.2 词法分析器设计

使用visual studio编辑.lex文件，对所设计的高级语言程序进行词法分析。

### 定义部分

1. 定义部分中引入相关头文件信息，以及定义需要使用的表达式和结构体等部分。在.lex文件中定义部分如下：

%{

#include "parser.tab.h"

#include "string.h"

#include "def.h"

int yycolumn=1;

int line=1;

#define YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; \

yylloc.first\_column=yycolumn; yylloc.last\_column=yycolumn+yyleng-1; yycolumn+=yyleng;

typedef union {

int type\_int;

char type\_char;

float type\_float;

char type\_bool[5];

char type\_string[1024];

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

%}

1. 且在定义部分中会定义一些后续可能会用到的正则表达式，即给这些正则表达式取一个别名。定义如下：

%option yylineno

%x COMMENT

/\*标识符\*/

ID {Fwords}{Mwords}\*

Fwords [a-zA-Z]|"\_"

Mwords {Fwords}|Digit

Digit [0-9]

/\*整型常量\*/

INT {Digit}+|{Fnum}{Hnum}+

Fnum ("0x"|"0X")

Hnum {Digit}|[a-fA-F]

/\*浮点型常量\*/

FLOAT [0-9]\*[.][0-9]+([eE][-+]?([1-9][0-9\*]|[0]))?

/\*单字符常量\*/

CHAR ['][^\n][']

/\*字符串常量\*/

STRING ["][^\n]\*["]

/\*布尔常量\*/

BOOL ("true"|"false")

/\*变量类型\*/

type ("bool"|"void"|"char"|"int"|"string"|"float")

/\*比较运算符\*/

compare ("<"|">"|"<="|">="|"=="|"!=")

ERRORID [0-9][A-Za-z0-9]\*

### 规则部分

规则部分是由正则表达式和相应的响应函数组成。规则部分代码如下：

%%

{INT} {printf("line%d:(整型常量,%s)\n",line,yytext);yylval.type\_int=atoi(yytext); return INT;}

{FLOAT} {printf("line%d:(浮点常量,%s)\n",line,yytext);yylval.type\_float=atof(yytext); return FLOAT;}

{CHAR} {printf("line%d:(单字符常量,%s)\n",line,yytext);yylval.type\_char=yytext[1];return CHAR;}

{STRING} {printf("line%d:(字符串常量,%s)\n",line,yytext);strncpy(yylval.type\_id,yytext+1,strlen(yytext)-2);return STRING;}//string

{BOOL} {printf("line%d:(布尔常量,%s)\n",line,yytext);strcpy(yylval.type\_bool, yytext);return BOOL;}

{type} {printf("line%d:(类型,%s)\n",line,yytext);strcpy(yylval.type\_id, yytext);return TYPE;}

"return" {printf("line%d:(函数返回,%s)\n",line,yytext);return RETURN;}

"if" {printf("line%d:(条件判断,%s)\n",line,yytext);return IF;}

"else" {printf("line%d:(条件判断,%s)\n",line,yytext);return ELSE;}

"while" {printf("line%d:(while循环,%s)\n",line,yytext);return WHILE;}

"break" {printf("line%d:(跳出循环,%s)\n",line,yytext);return BREAK;} // break

"continue" {printf("line%d:(继续循环,%s)\n",line,yytext);return CONTINUE;} //continue

"for" {printf("line%d:(循环,%s)\n",line,yytext);return FOR;}//for

"switch" {printf("line%d:(switch分支,%s)\n",line,yytext);return SWITCH;}//switch

"case" {printf("line%d:(case语句,%s)\n",line,yytext);return CASE;}//case

"struct" {printf("line%d:(结构体,%s)\n",line,yytext);return STRUCT;}//struct

"default" {printf("line%d:(default语句,%s)\n",line,yytext);return DEFAULT;}//DEFAULT

{ID} {printf("line%d:(变量,%s)\n",line,yytext);strcpy(yylval.type\_id, yytext); return ID;}

{ERRORID} {printf("line%d:(错误的变量定义,%s)\n",line,yytext);strcpy(yylval.type\_id,yytext);return ERRORID;}//错误标识符

":" {printf("line%d:(冒号,%s)\n",line,yytext);return COLON;}

";" {printf("line%d:(分号,%s)\n",line,yytext);return SEMI;}

"," {printf("line%d:(逗号,%s)\n",line,yytext);return COMMA;}

{compare} {printf("line%d:(比较运算,%s)\n",line,yytext);strcpy(yylval.type\_id, yytext);return RELOP;}

"=" {printf("line%d:(赋值运算,%s)\n",line,yytext);return ASSIGNOP;}

"+" {printf("line%d:(求和运算,%s)\n",line,yytext);return PLUS;}

"-" {printf("line%d:(求差运算,%s)\n",line,yytext);return MINUS;}

"\*" {printf("line%d:(求积运算,%s)\n",line,yytext);return STAR;}

"/" {printf("line%d:(求商运算,%s)\n",line,yytext);return DIV;}

"." {printf("line%d:(点号,%s)\n",line,yytext);return DOT;}

"%" {printf("line%d:(模运算,%s)\n",line,yytext);return MOD;}

"&&" {printf("line%d:(与运算,%s)\n",line,yytext);return AND;}

"||" {printf("line%d:(或运算,%s)\n",line,yytext);return OR;}

"!" {printf("line%d:(取非,%s)\n",line,yytext);return NOT;}

"++" {printf("line%d:(自增,%s)\n",line,yytext); return AUTOPLUS;}//自增

"--" {printf("line%d:(自减,%s)\n",line,yytext); return AUTOMINUS;}//自减

"+=" {printf("line%d:(求和赋值,%s)\n",line,yytext); return PLUSASSIGNOP;}

"-=" {printf("line%d:(求差赋值,%s)\n",line,yytext); return MINUSASSIGNOP;}

"\*=" {printf("line%d:(求积赋值,%s)\n",line,yytext);return STARASSIGNOP;}

"/=" {printf("line%d:(求商赋值,%s)\n",line,yytext);return DIVASSIGNOP;}

"%=" {printf("line%d:(求模赋值,%s)\n",line,yytext);return MODASSIGNOP;}

"(" {printf("line%d:(左括号小,%s)\n",line,yytext);return LP;}

")" {printf("line%d:(右括号小,%s)\n",line,yytext);return RP;}

"{" {printf("line%d:(左括号大,%s)\n",line,yytext);return LC;}

"}" {printf("line%d:(右括号大,%s)\n",line,yytext);return RC;}

"[" {printf("line%d:(左括号中,%s)\n",line,yytext);return LB;}

"]" {printf("line%d:(右括号中,%s)\n",line,yytext);return RB;}

[\n] {printf("line%d:(换行符,\\n)\n",line);yycolumn=1;line++;}

[ \r\t] {}

. {printf("Error type A :Mysterious character \"%s\"\n\t at Line %d\n",yytext,yylineno);}

\/\/[^\n]\* {printf("(%s, LINECOMMENT)\n", yytext);}//匹配注释

/\* 注释 \*/

"/\*" {BEGIN(COMMENT);}

<COMMENT>"\*/" {BEGIN(INITIAL);}

<COMMENT>([^\*]|\n)+|.

<COMMENT><<EOF>> {printf("%s: %d: Unterminated comment\n", yytext, yylineno);return 0;}

其中的yytext的类型未char\*，式flex提供的一个变量，里面保存了当前词法单元对应的词。

### 用户自定义代码部分

这部分代码会代码会被原封不动的拷贝到lex.yy.c中，以方便用户自定义所需要执行的函数（包括之前的main函数）。以下是用户子程序部分：

%%

//用户函数部分

int yywrap()

{

system("pause");

return 1;

}

## 2.3 词法分析器实现结果

1. 在终端输入如图 2-1指令，得到parser.exe程序

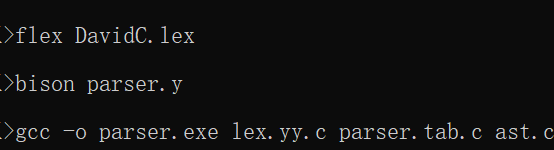


图 2-1 终端指令

1. 对simple.c进行分词测试，得到输出结果如图 2-2所示：

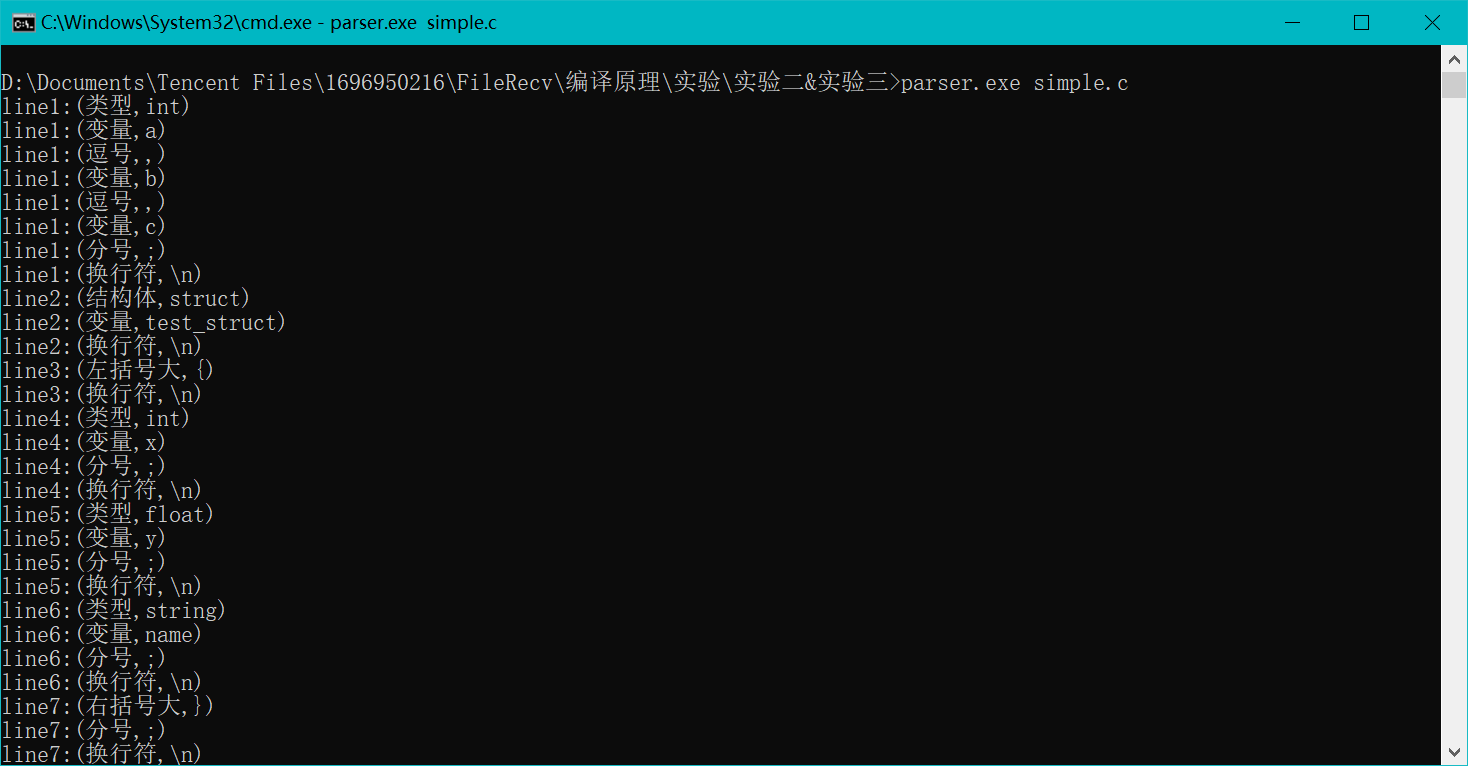


图 2-2 词法分析器运行结果

# 3 实验三 自定义语言语法分析器设计与实现

## 语法文法描述

1. 首先定义非终结符的类型，结合bison的语法规则，用%type定义非终结符的语义值类型。定义的非终结符如下：

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Exp\_less Args

%type <ptr> CaseStmtList CaseStmt DefaultStmt Struct\_dec StructName

%type <ptr> ForDec ForExp1 ForExp2 ForExp3

%type <ptr> Arraylist

1. 其次利用%token定义终结符的语义值类型。定义的终结符如下：

%token <type\_char> CHAR

%token <type\_string> STRING /\*指定STRING的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_bool> BOOL

%token <type\_int> INT /\*指定INT的语义值是type\_int，有词法分析得到的数值\*/

%token <type\_id> ID ERRORID RELOP TYPE /\*指定ID,RELOP,TYPE,STRING 的语义值是type\_id，有词法分析得到的标识符字符串\*/

%token <type\_float> FLOAT /\*指定FLOAT的语义值是type\_float，有词法分析得到的标识符字符串\*/

%token STRUCT RETURN FOR SWITCH CASE COLON DEFAULT

%token STRUCT\_VISIT STRUCT\_NEW STRUCT\_DEC EXT\_STRUCT\_DEC/\*结构体\*/

%token CASE\_STMT CASE\_STMT\_LIST DEFAULT\_STMT

%token FOR\_DEC FOR\_EXP1 FOR\_EXP2 FOR\_EXP3

%token DPLUS LP RP LC RC LB RB SEMI COMMA DOT /\*用bison对该文件编译时，带参数-d，生成的.tab.h中给这些单词进行编码，可在lex.l中包含parser.tab.h使用这些单词种类码\*/

%token PLUS MINUS STAR DIV MOD ASSIGNOP PLUSASSIGNOP MINUSASSIGNOP STARASSIGNOP DIVASSIGNOP MODASSIGNOP AND OR NOT AUTOPLUS AUTOMINUS IF ELSE WHILE BREAK CONTINUE /\*以下为接在上述token后依次编码的枚举常量，作为AST结点类型标记\*/

%token EXT\_DEF\_LIST EXT\_VAR\_DEF FUNC\_DEF FUNC\_DEC EXT\_DEC\_LIST PARAM\_LIST PARAM\_DEC VAR\_DEF DEC\_LIST DEF\_LIST COMP\_STM STM\_LIST EXP\_STMT IF\_THEN IF\_THEN\_ELSE

%token FUNC\_CALL ARGS FUNCTION PARAM ARG CALL LABEL GOTO JLT JLE JGT JGE EQ NEQ

%token ARRAY\_LIST //加入Array

1. 然后，定义运算符的优先级与结合性，如表 3-1所示：

表 3-1 算符优先级与结合性

|  |  |  |
| --- | --- | --- |
| 优先级 | 结合性 | 符号 |
| 高 | 左 | “+=”，”-=" |
|  | 左 | “=” |
|  | 左 | “||” |
|  | 左 | “&&” |
|  | 左 | ">"|"<"|">="|"<="|"=="|"!=" |
|  | 左 | “+”, “-“ |
|  | 左 | “\*”，“/” |
| 低 | 右 | “！”，“++”，“—” |

1. 最后，定义语法规则部分，参考实验一的语法定义，进行相应的规则编写实现以及一些扩充。语法规则定义如下：

%%

program: ExtDefList {displayRoot();display($1,0);/\*semantic\_Analysis0($1);\*/} //显示语法树,语义分析

;

ExtDefList: {$$=NULL;}

| ExtDef ExtDefList {$$=mknode(2,EXT\_DEF\_LIST,yylineno,$1,$2);} //每一个EXTDEFLIST的结点，其第1棵子树对应一个外部变量声明或函数

;

ExtDef: Specifier ExtDecList SEMI {$$=mknode(2,EXT\_VAR\_DEF,yylineno,$1,$2);} //该结点对应一个外部变量声明

|Specifier FuncDec CompSt {$$=mknode(3,FUNC\_DEF,yylineno,$1,$2,$3);} //该结点对应一个函数定义

|Struct\_dec SEMI{$$=mknode(1, EXT\_STRUCT\_DEC, yylineno,$1);}

| error SEMI {$$=NULL;}

;

Specifier: TYPE {$$=mknode(0,TYPE,yylineno);strcpy($$->type\_id,$1);$$->type=!strcmp($1,"int")?INT:!strcmp($1,"string")?STRING:BOOL;}

;

ExtDecList: VarDec {$$=$1;} /\*每一个EXT\_DECLIST的结点，其第一棵子树对应一个变量名(ID类型的结点),第二棵子树对应剩下的外部变量名\*/

| VarDec COMMA ExtDecList {$$=mknode(2,EXT\_DEC\_LIST,yylineno,$1,$3);}

;

VarDec: ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);} | //ID结点，标识符符号串存放结点的type\_id

ID Arraylist {$$=mknode(1,ARRAY\_LIST,yylineno,$2);strcpy($$->type\_id,$1);}

;

Arraylist: LB INT RB {$$=mknode(0,ARRAY\_LIST,yylineno);$$->type\_int=$2;}

| LB INT RB Arraylist {$$=mknode(2,ARRAY\_LIST,yylineno,$2,$4);$$->type\_int=$2;}

;

FuncDec: ID LP VarList RP {$$=mknode(1,FUNC\_DEC,yylineno,$3);strcpy($$->type\_id,$1);}//函数名存放在$$->type\_id

|ID LP RP {$$=mknode(0,FUNC\_DEC,yylineno);strcpy($$->type\_id,$1);$$->ptr[0]=NULL;}//函数名存放在$$->type\_id

;

VarList: ParamDec {$$=mknode(1,PARAM\_LIST,yylineno,$1);}

| ParamDec COMMA VarList {$$=mknode(2,PARAM\_LIST,yylineno,$1,$3);}

;

ParamDec: Specifier VarDec {$$=mknode(2,PARAM\_DEC,yylineno,$1,$2);}

;

CompSt: LC DefList StmList RC {$$=mknode(2,COMP\_STM,yylineno,$2,$3);}

;

StmList: {$$=NULL; }

| Stmt StmList {$$=mknode(2,STM\_LIST,yylineno,$1,$2);}

;

Stmt: Exp SEMI {$$=mknode(1,EXP\_STMT,yylineno,$1);}

| CompSt {$$=$1;} //复合语句结点直接最为语句结点，不再生成新的结点

| RETURN Exp SEMI {$$=mknode(1,RETURN,yylineno,$2);}

| IF LP Exp RP Stmt %prec LOWER\_THEN\_ELSE {$$=mknode(2,IF\_THEN,yylineno,$3,$5);}

| IF LP Exp RP Stmt ELSE Stmt {$$=mknode(3,IF\_THEN\_ELSE,yylineno,$3,$5,$7);}

| WHILE LP Exp RP Stmt {$$=mknode(2,WHILE,yylineno,$3,$5);}

| CONTINUE SEMI { $$ = mknode(0, CONTINUE, yylineno); strcpy($$->type\_id, "CONTINUE"); }

| BREAK SEMI { $$ = mknode(0, BREAK, yylineno); strcpy($$->type\_id, "BREAK"); }

| FOR LP ForDec RP Stmt {$$=mknode(2,FOR,yylineno,$3,$5);} //FOR循环识别 for(FORDEC):Stmt

| SWITCH LP Exp RP LC CaseStmtList RC {$$=mknode(2, SWITCH, yylineno, $3, $6);} //SWITCH识别

;

//用于识别Default:\*\*\*\*\*

DefaultStmt: DEFAULT COLON StmList {$$=mknode(1, DEFAULT\_STMT, yylineno, $3);}

;

//Case(INT): Case(CHAR):的识别

CaseStmt: CASE INT COLON StmList {$$=mknode(1, CASE\_STMT, yylineno, $4);$$->type\_int=$2;$$->type=INT;}

| CASE CHAR COLON StmList {$$=mknode(1, CASE\_STMT, yylineno, $4);$$->type\_char=$2;$$->type=CHAR;}

;

//用于识别连续多个case+Default(DEFAULT为终结符)

CaseStmtList:

{$$=NULL;}

| CaseStmt CaseStmtList {$$=mknode(2, CASE\_STMT\_LIST, yylineno, $1, $2);}

| DefaultStmt {$$=mknode(1, CASE\_STMT\_LIST, yylineno,$1);}

;

//用于识别For循环

//for(int i ; i < 10 ;i++)

//for(int i;i<10;)

//for(int i;;i++)

//for(int i;;)

//for(;i<10;i++)

//for(;i<10;)

//for(;;i++)

//for(;;)死循环

ForDec: ForExp1 SEMI ForExp2 SEMI ForExp3 {$$=mknode(3, FOR\_DEC,yylineno,$1,$3,$5);}

| ForExp1 SEMI SEMI ForExp3 {$$=mknode(3, FOR\_DEC,yylineno,$1,NULL,$4);}

| ForExp1 SEMI ForExp2 SEMI {$$=mknode(3, FOR\_DEC,yylineno,$1,$3,NULL);}

| ForExp1 SEMI SEMI {$$=mknode(3, FOR\_DEC,yylineno,$1,NULL,NULL);}

| SEMI ForExp2 SEMI ForExp3 {$$=mknode(3, FOR\_DEC,yylineno,NULL,$2,$4);}

| SEMI ForExp2 SEMI {$$=mknode(3, FOR\_DEC,yylineno,NULL,$2,NULL);}

| SEMI SEMI ForExp3 {$$=mknode(3, FOR\_DEC,yylineno,NULL,NULL,$3);}

| SEMI SEMI {$$=mknode(3, FOR\_DEC, yylineno, NULL, NULL, NULL);}

;

ForExp1: DecList {$$=mknode(1, FOR\_EXP1, yylineno, $1);}

| Specifier DecList {$$=mknode(2, FOR\_EXP1, yylineno, $1, $2);}

;

ForExp2: Exp {$$=mknode(1, FOR\_EXP2, yylineno, $1);}

;

ForExp3: Exp COMMA ForExp3 {$$=mknode(2, FOR\_EXP3, yylineno, $1,$3);}

| Exp {$$=mknode(2, FOR\_EXP3, yylineno, $1, NULL);}

;

//struct name {DefList}识别

//struct name ID识别

Struct\_dec: STRUCT StructName LC DefList RC {$$=mknode(2, STRUCT\_NEW, yylineno, $2, $4);}

| STRUCT ID ID {$$=mknode(0,STRUCT\_DEC,yylineno);strcpy($$->type\_id,$3);}

;

StructName: {$$=NULL;}

| ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}

;

DefList: {$$=NULL; }

| Def DefList {$$=mknode(2,DEF\_LIST,yylineno,$1,$2);}

| error SEMI {$$=NULL;}

;

Def: Specifier DecList SEMI {$$=mknode(2,VAR\_DEF,yylineno,$1,$2);}

|Struct\_dec SEMI {$$=$1;}

;

DecList: Dec {$$=mknode(1,DEC\_LIST,yylineno,$1);}

| Dec COMMA DecList {$$=mknode(2,DEC\_LIST,yylineno,$1,$3);}

;

Dec: VarDec {$$=$1;}

| VarDec ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}

;

Exp\_less: Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"ASSIGNOP");}//$$结点type\_id空置未用，正好存放运算符

| Exp PLUSASSIGNOP Exp {$$=mknode(2,PLUSASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "PLUSASSIGNOP");}

| Exp MINUSASSIGNOP Exp {$$=mknode(2,MINUSASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "MINUSASSIGNOP");}

| Exp STARASSIGNOP Exp {$$=mknode(2, STARASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"STARASSIGNOP");}

| Exp DIVASSIGNOP Exp {$$=mknode(2,DIVASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id,"DIVASSIGNOP");}

| Exp MODASSIGNOP Exp {$$=mknode(2,MODASSIGNOP,yylineno,$1,$3);strcpy($$->type\_id, "MODASSIGNOP");}

| Exp AND Exp {$$=mknode(2,AND,yylineno,$1,$3);strcpy($$->type\_id,"AND");}

| Exp OR Exp {$$=mknode(2,OR,yylineno,$1,$3);strcpy($$->type\_id,"OR");}

| Exp RELOP Exp {$$=mknode(2,RELOP,yylineno,$1,$3);strcpy($$->type\_id,$2);} //词法分析关系运算符号自身值保存在$2中

| Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type\_id,"PLUS");}

| Exp MINUS Exp {$$=mknode(2,MINUS,yylineno,$1,$3);strcpy($$->type\_id,"MINUS");}

| Exp STAR Exp {$$=mknode(2,STAR,yylineno,$1,$3);strcpy($$->type\_id,"STAR");}

| Exp DIV Exp {$$=mknode(2,DIV,yylineno,$1,$3);strcpy($$->type\_id,"DIV");}

| Exp MOD Exp {$$=mknode(2,MOD,yylineno,$1,$3);strcpy($$->type\_id,"MOD");}

| Exp AUTOPLUS {$$=mknode(1,AUTOPLUS,yylineno,$1);strcpy($$->type\_id, "RPAUTOPLUS");}

| Exp AUTOMINUS {$$=mknode(1,AUTOMINUS,yylineno,$1);strcpy($$->type\_id,"RPAUTOMINUS");}

| AUTOPLUS Exp {$$=mknode(1,AUTOPLUS,yylineno,$2);strcpy($$->type\_id, "LPAUTOPLUS");}

| AUTOMINUS Exp {$$=mknode(1,AUTOMINUS,yylineno,$2);strcpy($$->type\_id, "LPAUTOMINUS");}

| LP Exp RP {$$=$2;}

| MINUS Exp %prec UMINUS {$$=mknode(1,UMINUS,yylineno,$2);strcpy($$->type\_id,"UMINUS");}

| NOT Exp {$$=mknode(1,NOT,yylineno,$2);strcpy($$->type\_id,"NOT");}

| DPLUS Exp {$$=mknode(1,DPLUS,yylineno,$2);strcpy($$->type\_id,"DPLUS");}

| Exp DPLUS {$$=mknode(1,DPLUS,yylineno,$1);strcpy($$->type\_id,"DPLUS");}

| ID LP Args RP {$$=mknode(1,FUNC\_CALL,yylineno,$3);strcpy($$->type\_id,$1);}

| ID LP RP {$$=mknode(0,FUNC\_CALL,yylineno);strcpy($$->type\_id,$1);}

| ID {$$=mknode(0,ID,yylineno);strcpy($$->type\_id,$1);}

| CHAR {$$=mknode(0,CHAR,yylineno);$$->type\_char=$1;$$->type=CHAR;}

| STRING {$$=mknode(0,STRING,yylineno);strcpy($$->type\_id,$1);$$->type=STRING;}

| FLOAT {$$=mknode(0,FLOAT,yylineno);$$->type\_float=$1;$$->type=FLOAT;}

| BOOL {$$=mknode(0,BOOL,yylineno);strcpy($$->type\_bool,$1);;$$->type=BOOL;}

| ID LB Exp\_less RB {$$=mknode(1,ARRAY\_LIST,yylineno,$3);strcpy($$->type\_id,$1);} //数组

| ID Arraylist {$$=mknode(1,ID,yylineno,$2);strcpy($$->type\_id,$1);}

| Exp DOT ID {$$=mknode(1, STRUCT\_VISIT, yylineno, $1);strcpy($$->type\_id,$3);} //访问结构体元素

;

Exp: Exp\_less {$$=$1;}

| INT {$$=mknode(0,INT,yylineno);$$->type\_int=$1;$$->type=INT;}

;

Args: Exp COMMA Args {$$=mknode(2,ARGS,yylineno,$1,$3);}

| Exp {$$=mknode(1,ARGS,yylineno,$1);}

;

说明：对于如下语句语句

Exp:Exp PLUS Exp {$$=mknode(2,PLUS,yylineno,$1,$3);strcpy($$->type\_id,"PLUS");}

规则后面的{}中的是当完成归约时要执行的语义动作。规则做不的Exp的属性值用$$表示，右部有两个需要记录的属性值，即两个Exp，位置需要分别是1和3，故对应的属性值分别用$1和$3表示。

## 语法分析器设计

同.lex文件类似，.y文件也分为三个部分，即定义部分、规则部分、用户函数部分。

### 定义部分

所有的词法单元的定义都可以放到这部分。包括一些库函数的引用、外部变量的声明和函数声明。定义部分如下：

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

int yylex();

void yyerror(const char\* fmt, ...);

void display(struct ASTNode \*,int);

void displayRoot();

%}

此外，还有辅助定义部分，定义相关的联合体，如下：

%union {

int type\_int;

char type\_char;

float type\_float;

char type\_bool[5];

char type\_string[1024];

char type\_id[32];

struct ASTNode \*ptr;

};

### 规则部分

这部分包括具体的语法和相应的语义动作。具体来讲是在书写产生式。第一个产生式左边的非终结符默认为初始符号。产生式里的箭头在这里用冒号”:”表示，一组产生式与另一组之间以分号”；“隔开。产生式里无论是终结符还是非中介都都各自对应一个属性值，乘胜是左边的非终结符对应的属性值用$$表示，右边的几个符号的属性值按从左到右的顺序一次对应位$1、$2、$3等。每条产生式的最后可以添加一组以花括号”{“和”}”括起来的语义动作，这组语义动作会在整条产生式的最后可以添加一组产生式规约完成之后执行，如果不明确指定语义动作，那么bison将采用默认的语义动作{$$=$1}。需要注意的是，在产生式中间添加语义动作在某些情况下有可能在原有语法中引入冲突，因此能使用时要特别谨慎。具体的代码实现部分已经在3.1中体现，此处不再赘述。

### 用户函数部分

这部分代码会被原封不动的拷贝到parser.tab.c中，以方便用户自定义所需要的函数。用户自定义函数如下：

%%

int main(int argc, char \*argv[]){

yyin=fopen(argv[1],"r");

if (!yyin)

return 0;

yylineno=1;

yyparse();

return 0;

}

void yyerror(const char\* fmt, ...)

{

va\_list ap;

va\_start(ap, fmt);

fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first\_line,yylloc.first\_column);

vfprintf(stderr, fmt, ap);

fprintf(stderr, ".\n");

}

### 抽象语法树结点的建立

在语法分析阶段，要生成建立抽象语法树AST。抽象语法树将词法分析之后生成的单词元素都按照一定的规则组装起来，再利用树的的结构表示出文件中各语法元素的关系。AST树结点的定义如下：

struct ASTNode {

//以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

int kind;

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char;

char type\_bool[5];

char type\_string[1024];

};

struct ASTNode \*ptr[4]; //由kind确定有多少棵子树

int place; //存放（临时）变量在符号表的位置序号

char Etrue[15],Efalse[15]; //对布尔表达式的翻译时，真假转移目标的标号

char Snext[15]; //结点对应语句S执行后的下一条语句位置标号

struct codenode \*code; //该结点中间代码链表头指针

int type; //用以标识表达式结点的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //计数器，可以用来统计形参个数

};

### 显示抽象语法树

抽象语法树的遍历是树的先序遍历，将遍历的结果输出，对不同的结点输出的结果不同。Display函数设计如下：

void display(struct ASTNode \*T,int indent)

{//对抽象语法树的先根遍历

int i=1;

struct ASTNode \*T0;

if (T)

{

switch (T->kind)

{

case EXT\_DEF\_LIST:

display(T->ptr[0],indent); //显示该外部定义（外部变量和函数）列表中的第一个

display(T->ptr[1],indent); //显示该外部定义列表中的其它外部定义

break;

case EXT\_VAR\_DEF:

printf("%\*c外部变量定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+OFFSET); //显示外部变量类型

printf("%\*c变量名：\n",indent+OFFSET,' ');

display(T->ptr[1],indent+OFFSET\*2); //显示变量列表

break;

case TYPE:

printf("%\*c类型： %s\n",indent,' ',T->type\_id);

break;

case EXT\_DEC\_LIST:

display(T->ptr[0],indent); //依次显示外部变量名，

display(T->ptr[1],indent); //后续还有相同的，仅显示语法树此处理代码可以和类似代码合并

break;

case FUNC\_DEF:

printf("%\*c函数定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+OFFSET); //显示函数返回类型

display(T->ptr[1],indent+OFFSET); //显示函数名和参数

display(T->ptr[2],indent+OFFSET); //显示函数体

break;

case FUNC\_DEC:

printf("%\*c函数名：%s\n",indent,' ',T->type\_id);

if (T->ptr[0])

{

printf("%\*c函数形参：\n",indent,' ');

display(T->ptr[0],indent+OFFSET); //显示函数参数列表

}

else printf("%\*c无参函数\n",indent+OFFSET,' ');

break;

case PARAM\_LIST:

display(T->ptr[0],indent); //依次显示全部参数类型和名称，

display(T->ptr[1],indent);

break;

case PARAM\_DEC:

printf("%\*c类型：%s, 参数名：%s\n",indent,' ',T->ptr[0]->type==INT?"int":"float",T->ptr[1]->type\_id);

break;

case EXP\_STMT:

printf("%\*c表达式语句：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+OFFSET);

break;

case RETURN:

printf("%\*c返回语句：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+OFFSET);

break;

case COMP\_STM:

printf("%\*c复合语句：(%d)\n",indent,' ',T->pos);

printf("%\*c复合语句的变量定义部分：\n",indent+OFFSET,' ');

display(T->ptr[0],indent+OFFSET\*2); //显示定义部分

printf("%\*c复合语句的语句部分：\n",indent+OFFSET,' ');

display(T->ptr[1],indent+OFFSET\*2); //显示语句部分

break;

case STM\_LIST:

display(T->ptr[0],indent); //显示第一条语句

display(T->ptr[1],indent); //显示剩下语句

break;

case WHILE:

printf("%\*cWHILE循环(语句)：(%d)\n",indent,' ',T->pos);

printf("%\*c循环条件：\n",indent+OFFSET,' ');

display(T->ptr[0],indent+OFFSET\*2); //显示循环条件

printf("%\*c循环体：(%d)\n",indent+OFFSET,' ',T->pos);

display(T->ptr[1],indent+OFFSET\*2); //显示循环体

break;

case IF\_THEN:

printf("%\*c条件语句(IF\_THEN)：(%d)\n",indent,' ',T->pos);

printf("%\*c条件：\n",indent+OFFSET,' ');

display(T->ptr[0],indent+OFFSET\*2); //显示条件

printf("%\*cIF子句：(%d)\n",indent+OFFSET,' ',T->pos);

display(T->ptr[1],indent+OFFSET\*2); //显示if子句

break;

case IF\_THEN\_ELSE:

printf("%\*c条件语句(IF\_THEN\_ELSE)：(%d)\n",indent,' ',T->pos);

printf("%\*c条件：\n",indent+OFFSET,' ');

display(T->ptr[0],indent+OFFSET\*2); //显示条件

printf("%\*cIF子句：(%d)\n",indent+OFFSET,' ',T->pos);

display(T->ptr[1],indent+OFFSET\*2); //显示if子句

printf("%\*cELSE子句：(%d)\n",indent+OFFSET,' ',T->pos);

display(T->ptr[2],indent+OFFSET\*2); //显示else子句

break;

case DEF\_LIST:

display(T->ptr[0],indent); //显示该局部变量定义列表中的第一个

display(T->ptr[1],indent); //显示其它局部变量定义

break;

case VAR\_DEF:

printf("%\*c局部变量定义：(%d)\n",indent,' ',T->pos);

display(T->ptr[0],indent+OFFSET); //显示变量类型

display(T->ptr[1],indent+OFFSET); //显示该定义的全部变量名

break;

case DEC\_LIST:

printf("%\*c变量名：\n",indent,' ');

T0=T;

while (T0) {

if (T0->ptr[0]->kind==ID){

printf("%\*c %s\n",indent+OFFSET\*2,' ',T0->ptr[0]->type\_id);

}

else if (T0->ptr[0]->kind==ASSIGNOP)

{

if(T0->ptr[0]->ptr[0]->kind == ARRAY\_LIST) display(T0->ptr[0]->ptr[0],indent); //var\_dec assignop exp的情况

printf("%\*c %s 赋值语句\n ",indent+OFFSET\*2,' ',T0->ptr[0]->ptr[0]->type\_id);

display(T0->ptr[0]->ptr[1],indent+strlen(T0->ptr[0]->ptr[0]->type\_id)+7); //显示初始化表达式

}

else if(T0->ptr[0]->kind==ARRAY\_LIST){

printf("%\*c 数组%s\n",indent+OFFSET\*2,' ',T0->ptr[0]->type\_id);//var $$=$1

}

T0=T0->ptr[1];

}

break;

case ID:

printf("%\*cID： %s\n",indent,' ',T->type\_id);

break;

case CONTINUE:

printf("%\*c CONTINUE语句(%d)\n", indent, ' ', T->pos);

break;

case BREAK:

printf("%\*c BREAK语句(%d)\n", indent, ' ', T->pos);

break;

//数组的打印

case ARRAY\_LIST:

T0 = T;

while (T0) {

if (T0->ptr[0]->kind == ARRAY\_LIST) {

printf("%\*c 数组%s\n", indent + OFFSET \* 2, ' ', T0->ptr[0]->type\_id);//var $$=$1

}

else {

display(T0->ptr[0], indent + OFFSET);

}

T0 = T0->ptr[1];

}

break;

case INT:

printf("%\*cINT 整型常量：%d\n",indent,' ',T->type\_int);

break;

case FLOAT:

printf("%\*cFLAOT 浮点型常量：%f\n",indent,' ',T->type\_float);

break;

case CHAR:

printf("%\*cCHAR 单字符常量: %c\n", indent, ' ', T->type\_char);

break;

case BOOL:

printf("%\*cBOOL 布尔常量: %s\n", indent, ' ', T->type\_bool);

break;

case STRING:

printf("%\*cSTRING 字符串常量: %s\n", indent, ' ', T->type\_string);//STRING

break;

case ASSIGNOP:

T0=T;

if(T0->ptr[0]->kind==ARRAY\_LIST){

printf("%\*c数组%s\n",indent+OFFSET\*2,' ',T0->ptr[0]->type\_id);

}

case PLUSASSIGNOP:

case MINUSASSIGNOP:

case STARASSIGNOP:

case DIVASSIGNOP:

case MODASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case AUTOPLUS:

case AUTOMINUS:

case MINUS:

case STAR:

case DIV:

case MOD:

printf("%\*c%s\n",indent,' ',T->type\_id);

display(T->ptr[0],indent+OFFSET);

display(T->ptr[1],indent+OFFSET);

break;

case NOT:

case UMINUS:

printf("%\*c%s\n",indent,' ',T->type\_id);

display(T->ptr[0],indent+OFFSET);

break;

case FUNC\_CALL:

printf("%\*c函数调用：(%d)\n",indent,' ',T->pos);

printf("%\*c函数名：%s\n",indent+OFFSET,' ',T->type\_id);

display(T->ptr[0],indent+OFFSET);

break;

case ARGS:

i=1;

while (T) { //ARGS表示实际参数表达式序列结点，其第一棵子树为其一个实际参数表达式，第二棵子树为剩下的

struct ASTNode \*T0=T->ptr[0];

printf("%\*c第%d个：\n",indent,' ',i++);

display(T0,indent+OFFSET);

T=T->ptr[1];

}

// printf("%\*c第%d个实际参数表达式：\n",indent,' ',i);

// display(T,indent+OFFSET);

printf("\n");

break;

case SWITCH:

printf("%\*cSwitch语句:(%d)\n", indent, ' ',T->pos);

printf("%\*c判断对象:\n", indent+3, ' ');

display(T->ptr[0], indent+6);

display(T->ptr[1],indent+6);

break;

case CASE\_STMT\_LIST:

display(T->ptr[0], indent);

display(T->ptr[1], indent);

break;

case CASE\_STMT:

printf("%\*cCase语句:\n", indent, ' ');

printf("%\*c匹配值:\n", indent+3, ' ');

if (T->type == CHAR){

printf("%\*cCHAR：'%c'\n", indent+6, ' ', T->type\_char);

} else if (T->type == INT) {

printf("%\*cINT：%d\n", indent+6, ' ', T->type\_int);

}

printf("%\*cCase语句块:\n", indent+3, ' ');

display(T->ptr[0], indent+6);

break;

case DEFAULT\_STMT:

printf("%\*cDefualt语句:\n", indent, ' ');

printf("%\*cDefault语句块:\n", indent+3, ' ');

display(T->ptr[0], indent+6);

break;

case STRUCT\_NEW:

printf("%\*c定义结构体:\n", indent, ' ');

display(T->ptr[0], indent+6);

printf("%\*c结构体成员:\n", indent, ' ');

display(T->ptr[1], indent+6);

break;

case STRUCT\_DEC:

printf("%\*c新建结构体变量:\n", indent, ' ');

printf("%\*c 变量名:%s\n", indent, ' ', T->type\_id);

break;

case EXT\_STRUCT\_DEC:

printf("%\*c外部结构体:\n", indent, ' ');

display(T->ptr[0], indent+3);

break;

case STRUCT\_VISIT:

printf("%\*c访问对象:\n", indent, ' ');

display(T->ptr[0], indent+3);

printf("%\*c目标元素:\n", indent, ' ');

printf("%\*cID: %s\n",indent+3, ' ', T->type\_id);

break;

case FOR:

printf("%\*cFor(循环)：(%d)\n", indent, ' ', T->pos);

printf("%\*c循环条件：\n", indent + 3, ' ');

display(T->ptr[0], indent + 6); //显示循环条件

printf("%\*c循环体：(%d)\n", indent + 3, ' ', T->pos);

display(T->ptr[1], indent + 6); //显示循环体

break;

case FOR\_DEC:

display(T->ptr[0], indent + 6);

display(T->ptr[1], indent + 6);

display(T->ptr[2], indent + 6);

break;

case FOR\_EXP1:

printf("%\*c循环初始语句(%d)\n", indent, ' ', T->pos);

display(T->ptr[0], indent + 3);

display(T->ptr[1], indent + 3);

break;

case FOR\_EXP2:

printf("%\*c循环条件语句(%d)\n", indent, ' ', T->pos);

display(T->ptr[0], indent + 3);

break;

case FOR\_EXP3:

printf("%\*c循环结束条件(%d)\n", indent, ' ', T->pos);

while (T)

{

struct ASTNode \*T0 = T->ptr[0];

display(T0, indent + 3);

T = T->ptr[1];

}

break;

default:

printf("未找到该类型\n");

break;

}

}

}

## 语法分析器实现结果

1. 在终端输入如图 2-1指令，得到parser.exe程序。
2. 对simple.c进行分词测试，得到输出结果如图 3-1所示



图 3-1 语法分析器运行结果

# 4 实验四 符号表管理和属性计算

## 4.1 符号表结构

符号表在编译程序工作的过程中需要不断收集、记录和使用源程序中一些语法符号的类型和特征等相关信息。这些信息一般以表格形式存储于系统中。如常数表、变量名表、数组名表、过程名表、标号表等等，统称为符号表。对于符号表组织、构造和管理方法的好坏会直接影响编译系统的运行效率。

符号表采用顺序表进行管理，用单表实现，用一个符号栈老表示在当前作用域内的付哈，每当有一个新的符号出现，则将新的符号以及对应的属性压入符号栈中。当作用域结束之后就将退栈。

符号表代码实现如下：

struct symbol { //这里只列出了一个符号表项的部分属性，没考虑属性间的互斥

char name[33]; //变量或函数名

int level; //层号

int type; //变量类型或函数返回值类型

int paramnum; //对函数适用，记录形式参数个数

char alias[10]; //别名，为解决嵌套层次使用

char flag; //符号标记，函数：'F' 变量：'V' 参数：'P' 临时变量：'T'

char offset; //外部变量和局部变量在其静态数据区或活动记录中的偏移量，

//或记录函数活动记录大小，目标代码生成时使用

//函数入口等实验可能会用到的属性...

};

//符号表

struct symboltable{

struct symbol symbols[MAXLENGTH];

int index;

} symbolTable;

struct symbol\_scope\_begin {

//当前作用域的符号在符号表的起始位置序号,这是一个栈结构,当使用顺序表作为符号表时，进入、退出一个作用域时需要对其操作，以完成符号表的管理。对其它形式的符号表，不一定需要此数据结构

int TX[30];

int top;

} symbol\_scope\_TX;

## 4.2 符号表显示

（1）外部变量在符号表中的显示。

外部变量在符号表中的起始序号为0，代码如下：

//查符号表，获得符号表中的位置，类型送type

rtn = searchSymbolTable(T->type\_id);

if (rtn == -1)

semantic\_error(T->pos, T->type\_id, "变量未定义");

if (symbolTable.symbols[rtn].flag == 'F')

semantic\_error(T->pos, T->type\_id, "是函数名，类型不匹配");

/\*else if (symbolTable.symbols[rtn].flag == 'A')

semantic\_error(T->pos, T->type\_id, "是数组变量,类型不匹配");\*/

else {

T->place = rtn; //结点保存变量在符号表中的位置

T->code = NULL; //标识符不需要生成TAC

T->type = symbolTable.symbols[rtn].type;

T->offset = symbolTable.symbols[rtn].offset;

T->width = 0; //未再使用新单元

}

（2）临时变量在符号表中的显示：

当存在有需要用到临时变量的表达式，例如a+b+c，a+b的结果需要保存到一个临时变量t中，然后才能计算t+c的值。实现临时变量的符号表显示的代码如下：

//填写临时变量到符号表，返回临时变量在符号表中的位置

int fill\_Temp(char\* name, int level, int type, char flag, int offset) {

strcpy(symbolTable.symbols[symbolTable.index].name, "");

strcpy(symbolTable.symbols[symbolTable.index].alias, name);

symbolTable.symbols[symbolTable.index].level = level;

symbolTable.symbols[symbolTable.index].type = type;

symbolTable.symbols[symbolTable.index].flag = flag;

symbolTable.symbols[symbolTable.index].offset = offset;

return symbolTable.index++; //返回的是临时变量在符号表中的位置序号

}

1. 填写符号表：

//填符号表

int fillSymbolTable(char\* name, char\* alias, int level, int type, char flag, int offset) {

//首先根据name查符号表，不能重复定义 重复定义返回-1

int i;

/\*符号查重，考虑外部变量声明前有函数定义，

其形参名还在符号表中，这时的外部变量与前函数的形参重名是允许的\*/

for (i = symbolTable.index - 1; i >= 0 && (symbolTable.symbols[i].level == level || level == 0); i--) {

if (level == 0 && symbolTable.symbols[i].level == 1)

continue; //外部变量和形参不必比较重名

if (!strcmp(symbolTable.symbols[i].name, name))

return -1;

}

//填写符号表内容

strcpy(symbolTable.symbols[symbolTable.index].name, name);

strcpy(symbolTable.symbols[symbolTable.index].alias, alias);

symbolTable.symbols[symbolTable.index].level = level;

symbolTable.symbols[symbolTable.index].type = type;

symbolTable.symbols[symbolTable.index].flag = flag;

symbolTable.symbols[symbolTable.index].offset = offset;

return symbolTable.index++; //返回的是符号在符号表中的位置序号，中间代码生成时可用序号取到符号别名

}

## 4.3 属性计算

即需要计算每个变量在实际存储中的地址偏移量。

规定：int类型占4字节、float类型占8字节、char和bool类型占1字节。

则顺序计算每一个变量的地址偏移量时，只需要在前一个变量地址的基础上加上前一个变量类型所占有的字节数即可。代码实现如下：

case INT:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为整常量生成一个临时变量

T->type = INT;

opn1.kind = INT;

opn1.const\_int = T->type\_int;

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->place].alias);

result.offset = symbolTable.symbols[T->place].offset;

T->code = genIR(ASSIGNOP, opn1, opn2, result);

T->width = 4;

break;

case FLOAT:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为浮点常量生成一个临时变量

T->type = FLOAT;

opn1.kind = FLOAT;

opn1.const\_float = T->type\_float;

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->place].alias);

result.offset = symbolTable.symbols[T->place].offset;

T->code = genIR(ASSIGNOP, opn1, opn2, result);

T->width = 8;

break;

case CHAR:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset);

//为字符常量生成一个临时变量

T->type = CHAR;

opn1.kind = CHAR;

opn1.const\_char = T->type\_char;

result.kind = ID;

result.offset = symbolTable.symbols[T->place].offset;

T->code = genIR(ASSIGNOP, opn1, opn2, result);

T->width = 1;

break;

case BOOL:

T->place = fill\_Temp(newTemp(), LEV, T->type, 'T', T->offset); //为浮点常量生成一个临时变量

T->type = BOOL;

opn1.kind = BOOL;

strcpy(opn1.const\_bool, T->type\_bool);

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->place].alias);

result.offset = symbolTable.symbols[T->place].offset;

T->code = genIR(ASSIGNOP, opn1, opn2, result);

T->width = 1;

break;

数组类型变量只需要获得数组数据的大小，再乘上数组类型占用的字节数，即可计算出数组所占用的字节数。

## 4.4 符号表管理和属性计算结果

通过如图 4-1所示的指令，对相关文件进行编译后得到parser.exe程序

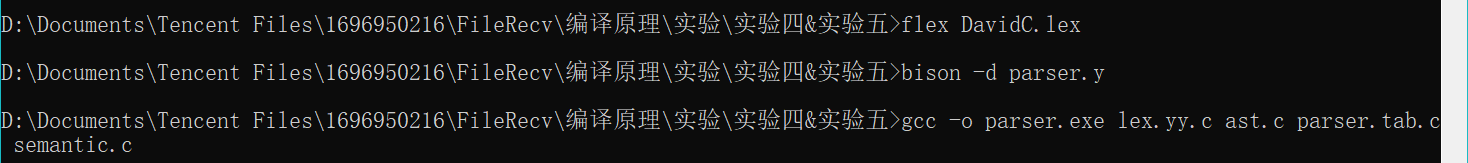


图 4-1 终端指令

然后用所得到的parser.exe对simple.c进行测试，得到结果如图 4-2所示：

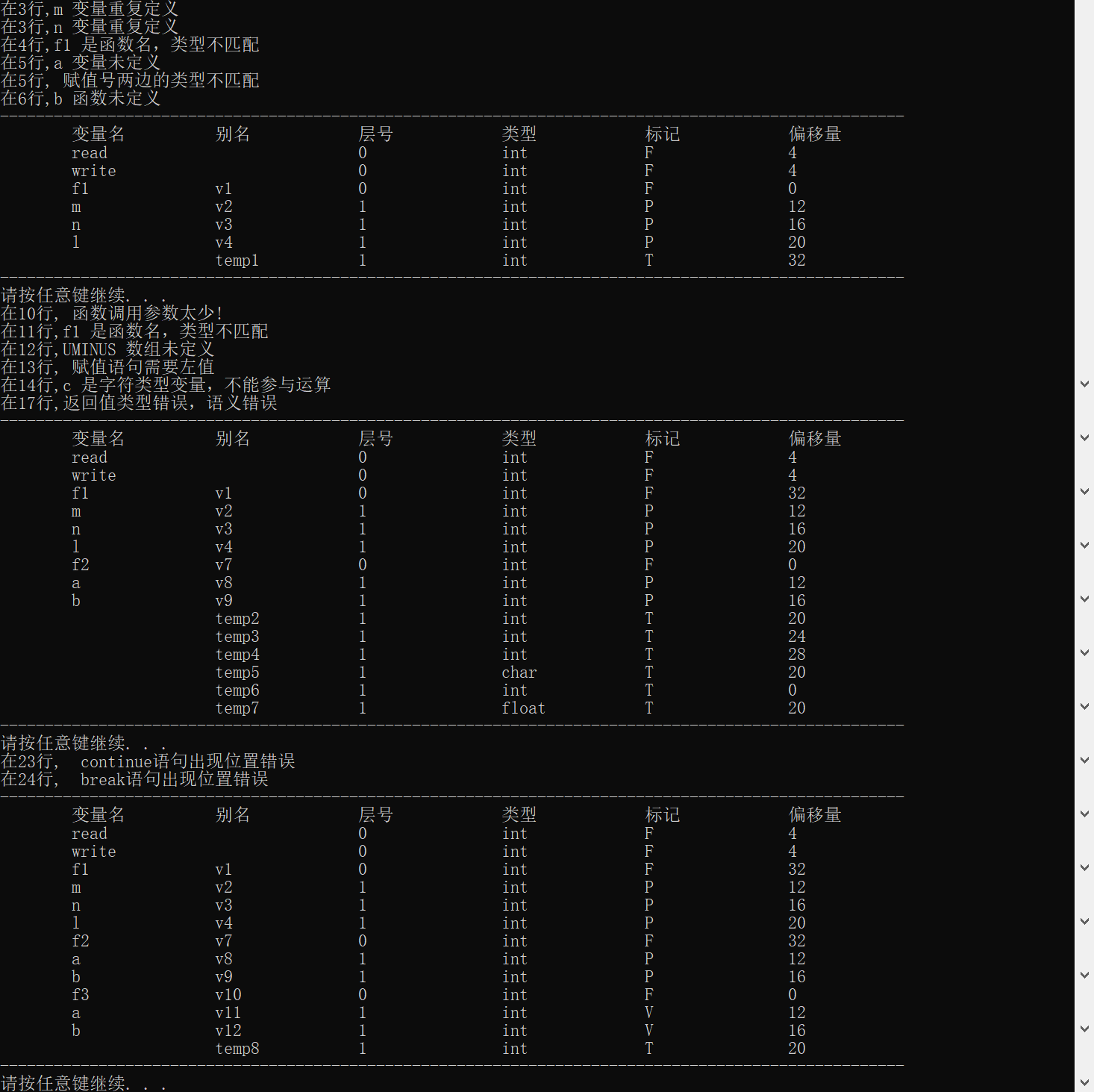


图 4-2 符号表测试结果

# 5 实验五 静态语义分析

## 错误类型定义

（1）错误类型1：使用未定义的变量；

（2）错误类型2：调用未定义或未声明的函数；

（3）错误类型3：变量重复定义；

（4）错误类型4：函数重复定义；

（5）错误类型5：对非函数名采用函数调用形式；

（6）错误类型6：对函数名采用非函数调用形式访问；

（7）错误类型7：函数调用时参数个数不匹配，如实参表达式个数太多、或实参表达式个数太少；

（8）错误类型8：函数调用时实参和形参类型不匹配；

（9）错误类型9：对非数组变量采用下标变量的形式访问；

（10）错误类型10：数组变量的下标不是整型表达式；

（11）错误类型11：赋值号左边不是左值表达式；

（12）错误类型12：对非左值表达式进行自增、自减运算；

（13）错误类型13：操作数类型不匹配或操作数类型域操作符不匹配（例如整形变量域数组变量相加减，或数组（或结构体）变量域数组（或结构体）结构体变量相加减）；

（14）错误类型14：函数返回值类型与函数定义的返回值类型不匹配；

（15）错误类型15：break语句不在循环语句中；

（16）错误类型16：continue语句不在循环语句中；

## 语义分析实现

函数semantic\_Analysis为静态语义分析函数，在此之前还定义了一个semantic\_Analysis0函数来实现准备工作并对semantic\_Analysis进行调用，具体实现如下：

void semantic\_Analysis(struct ASTNode\* T)

{//对抽象语法树的先根遍历,按display的控制结构修改完成符号表管理和语义检查和TAC生成（语句部分）

int rtn, num, width;

struct ASTNode\* T0;

struct opn opn1, opn2, result;

if (T)

{

switch (T->kind) {

case EXT\_DEF\_LIST:

if (!T->ptr[0])

break;

// 语义分析之前设置偏移地址

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //访问外部定义列表中的第一个

T->code = T->ptr[0]->code;

if (T->ptr[1]) {

T->ptr[1]->offset = T->ptr[0]->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //访问该外部定义列表中的其它外部定义

T->code = merge(2, T->code, T->ptr[1]->code);

}

break;

case EXT\_VAR\_DEF:

//处理外部说明,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

if (!strcmp(T->ptr[0]->type\_id, "int")) {

T->type = T->ptr[1]->type = INT;

T->ptr[1]->width = 4;

}

if (!strcmp(T->ptr[0]->type\_id, "float")) {

T->type = T->ptr[1]->type = FLOAT;

T->ptr[1]->width = 8;

}

if (!strcmp(T->ptr[0]->type\_id, "char")) {

T->type = T->ptr[1]->type = CHAR;

T->ptr[1]->width = 1;

}

if (!strcmp(T->ptr[0]->type\_id, "bool")) {

T->type = T->ptr[1]->type = BOOL;

T->ptr[1]->width = 2;

}

if (!strcmp(T->ptr[0]->type\_id, "string")) {

T->type = T->ptr[1]->type = STRING;

T->ptr[1]->width = 8;

}

T->ptr[1]->offset = T->offset; //这个外部变量的偏移量向下传递

ext\_var\_list(T->ptr[1]); //处理外部变量说明中的标识符序列

T->width = (T->ptr[1]->width) \* T->ptr[1]->num;//计算这个外部变量说明的宽度

T->code = NULL; //这里假定外部变量不支持初始化

break;

case FUNC\_DEF: //填写函数定义信息到符号表

if (!strcmp(T->ptr[0]->type\_id, "int")) {

T->ptr[1]->type = INT;

}

if (!strcmp(T->ptr[0]->type\_id, "float")) {

T->ptr[1]->type = FLOAT;

}

if (!strcmp(T->ptr[0]->type\_id, "char")) {

T->ptr[1]->type = CHAR;

}

T->width = 0; //函数的宽度设置为0，不会对外部变量的地址分配产生影响

T->offset = DX; //设置局部变量在活动记录中的偏移量初值

semantic\_Analysis(T->ptr[1]); //处理函数名和参数结点部分，这里不考虑用寄存器传递参数

T->offset += T->ptr[1]->width; //用形参单元宽度修改函数局部变量的起始偏移量

T->ptr[2]->offset = T->offset;

strcpy(T->ptr[2]->Snext, newLabel()); //函数体语句执行结束后的位置属性

semantic\_Analysis(T->ptr[2]); //处理函数体结点

//计算活动记录大小,这里offset属性存放的是活动记录大小，不是偏移

symbolTable.symbols[T->ptr[1]->place].offset = T->offset + T->ptr[2]->width;

T->code = merge(3, T->ptr[1]->code, T->ptr[2]->code, genLabel(T->ptr[2]->Snext)); //函数体的代码作为函数的代码

break;

case FUNC\_DEC: //根据返回类型，函数名填写符号表

rtn = fillSymbolTable(T->type\_id, newAlias(), LEV, T->type, 'F', 0);//函数不在数据区中分配单元，偏移量为0

if (rtn == -1) {

semantic\_error(T->pos, T->type\_id, "函数重复定义");

break;

}

else

T->place = rtn;

result.kind = ID;

strcpy(result.id, T->type\_id);

result.offset = rtn;

T->code = genIR(FUNCTION, opn1, opn2, result); //生成中间代码：FUNCTION 函数名

T->offset = DX; //设置形式参数在活动记录中的偏移量初值

if (T->ptr[0]) { //判断是否有参数

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理函数参数列表

T->width = T->ptr[0]->width;

symbolTable.symbols[rtn].paramnum = T->ptr[0]->num;

T->code = merge(2, T->code, T->ptr[0]->code); //连接函数名和参数代码序列

}

else

symbolTable.symbols[rtn].paramnum = 0, T->width = 0;

break;

case PARAM\_LIST: //处理函数形式参数列表

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

if (T->ptr[1]) {

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]);

T->num = T->ptr[0]->num + T->ptr[1]->num; //统计参数个数

T->width = T->ptr[0]->width + T->ptr[1]->width; //累加参数单元宽度

T->code = merge(2, T->ptr[0]->code, T->ptr[1]->code); //连接参数代码

}

else {

T->num = T->ptr[0]->num;

T->width = T->ptr[0]->width;

T->code = T->ptr[0]->code;

}

break;

case PARAM\_DEC:

rtn = fillSymbolTable(T->ptr[1]->type\_id, newAlias(), 1, T->ptr[0]->type, 'P', T->offset);

if (rtn == -1)

semantic\_error(T->ptr[1]->pos, T->ptr[1]->type\_id, "参数名重复定义");

else

T->ptr[1]->place = rtn;

T->num = 1; //参数个数计算的初始值

T->width = T->ptr[0]->type == INT ? 4 : 8; //参数宽度

result.kind = ID;

strcpy(result.id, symbolTable.symbols[rtn].alias);

result.offset = T->offset;

T->code = genIR(PARAM, opn1, opn2, result); //生成：FUNCTION 函数名

break;

case COMP\_STM:

LEV++;

//设置层号加1，并且保存该层局部变量在符号表中的起始位置在symbol\_scope\_TX

symbol\_scope\_TX.TX[symbol\_scope\_TX.top++] = symbolTable.index;

T->width = 0;

T->code = NULL;

if (T->ptr[0]) {

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理该层的局部变量DEF\_LIST

T->width += T->ptr[0]->width;

T->code = T->ptr[0]->code;

}

if (T->ptr[1]) {

T->ptr[1]->offset = T->offset + T->width;

strcpy(T->ptr[1]->Snext, T->Snext); //S.next属性向下传递

semantic\_Analysis(T->ptr[1]); //处理复合语句的语句序列

T->width += T->ptr[1]->width;

T->code = merge(2, T->code, T->ptr[1]->code);

}

//#if (DEBUG)

prn\_symbol(); //c在退出一个符合语句前显示的符号表

//system("pause");

//#endif

LEV--; //出复合语句，层号减1

symbolTable.index = symbol\_scope\_TX.TX[--symbol\_scope\_TX.top]; //删除该作用域中的符号

break;

case DEF\_LIST:

T->code = NULL;

if (T->ptr[0]) {

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]); //处理一个局部变量定义

T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

}

if (T->ptr[1]) {

T->ptr[1]->offset = T->offset + T->ptr[0]->width;

semantic\_Analysis(T->ptr[1]); //处理剩下的局部变量定义

T->code = merge(2, T->code, T->ptr[1]->code);

T->width += T->ptr[1]->width;

}

break;

case VAR\_DEF:

//处理一个局部变量定义,将第一个孩子(TYPE结点)中的类型送到第二个孩子的类型域

//类似于上面的外部变量EXT\_VAR\_DEF，换了一种处理方法

T->code = NULL;

if (!strcmp(T->ptr[0]->type\_id, "int")) {

T->ptr[1]->type = INT;

width = 4;

}

if (!strcmp(T->ptr[0]->type\_id, "float")) {

T->ptr[1]->type = FLOAT;

width = 8;

}

if (!strcmp(T->ptr[0]->type\_id, "char")) {

T->ptr[1]->type = CHAR;

width = 1;

}

if (!strcmp(T->ptr[0]->type\_id, "bool")) {

T->ptr[1]->type = BOOL;

width = 2;

}

if (!strcmp(T->ptr[0]->type\_id, "string")) {

T->ptr[1]->type = STRING;

width = 8;

}

T0 = T->ptr[1]; //T0为变量名列表子树根指针，对ID、ASSIGNOP类结点在登记到符号表，作为局部变量

num = 0;

T0->offset = T->offset;

T->width = 0;

while (T0) {

//处理所有DEC\_LIST结点

num++;

T0->ptr[0]->type = T0->type; //类型属性向下传递

if (T0->ptr[1])

T0->ptr[1]->type = T0->type;

T0->ptr[0]->offset = T0->offset; //类型属性向下传递

if (T0->ptr[1])

T0->ptr[1]->offset = T0->offset + width;

if (T0->ptr[0]->kind == ID) {

rtn = fillSymbolTable(T0->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'V', T->offset + T->width);//此处偏移量未计算，暂时为0

if (rtn == -1)

semantic\_error(T0->ptr[0]->pos, T0->ptr[0]->type\_id, "变量重复定义");

else T0->ptr[0]->place = rtn;

T->width += width;

}

else if (T0->ptr[0]->kind == ASSIGNOP) {

rtn = fillSymbolTable(T0->ptr[0]->ptr[0]->type\_id, newAlias(), LEV, T0->ptr[0]->type, 'V', T->offset + T->width);//此处偏移量未计算，暂时为0

if (rtn == -1)

semantic\_error(T0->ptr[0]->ptr[0]->pos, T0->ptr[0]->ptr[0]->type\_id, "变量重复定义");

else {

T0->ptr[0]->place = rtn;

T0->ptr[0]->ptr[1]->offset = T->offset + T->width + width;

Exp(T0->ptr[0]->ptr[1]);

opn1.kind = ID;

strcpy(opn1.id, symbolTable.symbols[T0->ptr[0]->ptr[1]->place].alias);

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T0->ptr[0]->place].alias);

T->code = merge(3, T->code, T0->ptr[0]->ptr[1]->code, genIR(ASSIGNOP, opn1, opn2, result));

}

T->width += width + T0->ptr[0]->ptr[1]->width;

}

T0 = T0->ptr[1];

}

break;

case STM\_LIST:

if (!T->ptr[0]) {

T->code = NULL;

T->width = 0;

break;

} //空语句序列

if (T->ptr[1]) //2条以上语句连接，生成新标号作为第一条语句结束后到达的位置

strcpy(T->ptr[0]->Snext, newLabel());

else //语句序列仅有一条语句，S.next属性向下传递

strcpy(T->ptr[0]->Snext, T->Snext);

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

if (T->ptr[1]) { //2条以上语句连接,S.next属性向下传递

strcpy(T->ptr[1]->Snext, T->Snext);

T->ptr[1]->offset = T->offset; //顺序结构共享单元方式

// T->ptr[1]->offset=T->offset+T->ptr[0]->width; //顺序结构顺序分配单元方式

semantic\_Analysis(T->ptr[1]);

//序列中第1条为表达式语句，返回语句，复合语句时，第2条前不需要标号

if (T->ptr[0]->kind == RETURN || T->ptr[0]->kind == EXP\_STMT || T->ptr[0]->kind == COMP\_STM)

T->code = merge(2, T->code, T->ptr[1]->code);

else

T->code = merge(3, T->code, genLabel(T->ptr[0]->Snext), T->ptr[1]->code);

if (T->ptr[1]->width > T->width) T->width = T->ptr[1]->width; //顺序结构共享单元方式

// T->width+=T->ptr[1]->width;//顺序结构顺序分配单元方式

}

break;

case IF\_THEN:

strcpy(T->ptr[0]->Etrue, newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse, T->Snext);

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

boolExp(T->ptr[0]);

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

T->code = merge(3, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue), T->ptr[1]->code);

break; //控制语句都还没有处理offset和width属性

case IF\_THEN\_ELSE:

strcpy(T->ptr[0]->Etrue, newLabel()); //设置条件语句真假转移位置

strcpy(T->ptr[0]->Efalse, newLabel());

T->ptr[0]->offset = T->ptr[1]->offset = T->ptr[2]->offset = T->offset;

boolExp(T->ptr[0]); //条件，要单独按短路代码处理

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, T->Snext);

semantic\_Analysis(T->ptr[1]); //if子句

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

strcpy(T->ptr[2]->Snext, T->Snext);

semantic\_Analysis(T->ptr[2]); //else子句

if (T->width < T->ptr[2]->width)

T->width = T->ptr[2]->width;

T->code = merge(6, T->ptr[0]->code, genLabel(T->ptr[0]->Etrue), T->ptr[1]->code, \

genGoto(T->Snext), genLabel(T->ptr[0]->Efalse), T->ptr[2]->code);

break;

case WHILE:

strcpy(T->ptr[0]->Etrue, newLabel()); //子结点继承属性的计算

strcpy(T->ptr[0]->Efalse, T->Snext);

T->ptr[0]->offset = T->ptr[1]->offset = T->offset;

boolExp(T->ptr[0]); //循环条件，要单独按短路代码处理

T->width = T->ptr[0]->width;

strcpy(T->ptr[1]->Snext, newLabel());

flag++; //修改成+1来控制层数

semantic\_Analysis(T->ptr[1]); //循环体

if (T->width < T->ptr[1]->width)

T->width = T->ptr[1]->width;

T->code = merge(5, genLabel(T->ptr[1]->Snext), T->ptr[0]->code, \

genLabel(T->ptr[0]->Etrue), T->ptr[1]->code, genGoto(T->ptr[1]->Snext));

flag--;

break;

case EXP\_STMT:

T->ptr[0]->offset = T->offset;

semantic\_Analysis(T->ptr[0]);

T->code = T->ptr[0]->code;

T->width = T->ptr[0]->width;

break;

case RETURN:

if (T->ptr[0]) {

T->ptr[0]->offset = T->offset;

Exp(T->ptr[0]);

num = symbolTable.index;

do

num--;

while (symbolTable.symbols[num].flag != 'F');

if (T->ptr[0]->type != symbolTable.symbols[num].type) {

semantic\_error(T->pos, "返回值类型错误，语义错误", "");

T->width = 0;

T->code = NULL;

return;

}

/\*需要判断返回值类型是否匹配\*/

T->width = T->ptr[0]->width;

result.kind = ID;

strcpy(result.id, symbolTable.symbols[T->ptr[0]->place].alias);

result.offset = symbolTable.symbols[T->ptr[0]->place].offset;

T->code = merge(2, T->ptr[0]->code, genIR(RETURN, opn1, opn2, result));

}

else {

T->width = 0;

result.kind = 0;

T->code = genIR(RETURN, opn1, opn2, result);

}

break;

case FOR:

LEV++;

flag++;

semantic\_Analysis(T->ptr[0]); //循环条件

semantic\_Analysis(T->ptr[1]); //循环体

flag--;

break;

case FOR\_DEC:

semantic\_Analysis(T->ptr[0]); //循环初始语句

semantic\_Analysis(T->ptr[1]); //循环条件

semantic\_Analysis(T->ptr[2]); //循环变换表达式

break;

case FOR\_EXP1:

Exp(T->ptr[0]);

Exp(T->ptr[1]);

break;

case FOR\_EXP2:

T0 = T;

while (T0) {

boolExp(T0->ptr[0]);

T0 = T0->ptr[1];

}

boolExp(T->ptr[0]);

break;

case FOR\_EXP3:

T0 = T;

while (T0) {

semantic\_Analysis(T0->ptr[0]);

T0 = T0->ptr[1];

}

break;

case BREAK:

if (flag == 0)

semantic\_error(T->pos, " ", "break语句出现位置错误");

break;

case CONTINUE:

if (flag == 0)

semantic\_error(T->pos, " ", "continue语句出现位置错误");

break;

case ID:

case INT:

case FLOAT:

case CHAR:

case ARRAY\_DF:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case PLUSASSIGNOP:

case MINUSASSIGNOP:

case DIVASSIGNOP:

case STARASSIGNOP:

case MODASSIGNOP:

case AUTOPLUS:

case AUTOMINUS:

case NOT:

case UMINUS:

case FUNC\_CALL:

case ARRAY\_CALL:

case ARGS:

Exp(T); //处理基本表达式

break;

}

}

}

void semantic\_Analysis0(struct ASTNode\* T) {

symbolTable.index = 0;

fillSymbolTable("read", "", 0, INT, 'F', 4);

symbolTable.symbols[0].paramnum = 0;//read的形参个数

fillSymbolTable("write", "", 0, INT, 'F', 4);

symbolTable.symbols[1].paramnum = 1;

//fillSymbolTable("x", "", 1, INT, 'P', 12);

symbol\_scope\_TX.TX[0] = 0; //外部变量在符号表中的起始序号为0

symbol\_scope\_TX.top = 1;

T->offset = 0; //外部变量在数据区的偏移量

semantic\_Analysis(T);

//prnIR(T->code);

//objectCode(T->code);

}

## 5.3 静态语义分析结果

如图 4-1所示得到parser.exe，在通过对simple.c进行测试，可以看出能对代码进行静态语义分析并找出对应的错误位置和错误信息。

静态语义分析显示的错误信息如图 5-1所示：

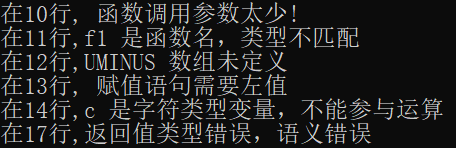
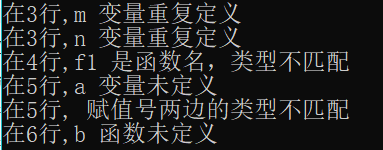


图 5-1 静态语义分析错误信息

# 6 实验六 中间代码生成

## 6.1 中间代码生成格式定义

选用四元式作为中间代码的形式，各种定义如表2-3所示：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| x[i] := y | 数组赋值 | ASSIGNARRAY | Y | I | X |
| x := y | 赋值操作 | ASSIGNOP | X |  | X |
| x := y + z | 加法操作 | PLUS | Y | Z | X |
| x := y - z | 减法操作 | MINUS | Y | Z | X |
| x := y \* z | 乘法操作 | STAR | Y | Z | X |
| x := y / z | 除法操作 | DIV | Y | Z | X |
| x := x + 1 | 自增操作 | AUTOADD | X | 1 | X |
| x := x - 1 | 自减操作 | AUTOSUB | X | 1 | X |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | F |
| PARAM x | 函数形参 | PARAM |  |  | X |
| LABEL x | 定义标号x | LABEL |  |  | X |
| GOTO x | 无条件转移 | GOTO |  |  | X |
| IF x [relop] y GOTO z | 条件转移 | [relop] | X | Y | Z |
| ARG x | 传实参x | ARG |  |  | X |
| CALL f | 调用函数 | CALL | F |  |  |
| PARAM x | 函数形参 | PARAM |  |  | X |
| LABEL x | 定义标号x | LABEL |  |  | X |
| BLOCK x | 定义基本块x | BLOCK |  |  | X |

## 6.2 中间代码生成翻译模式

### 1. 赋值语句和算数表达式的翻译模式

赋值语句和算数表达式的翻译模式如表 6-1所示：

表 6-1 赋值语句和算数表达式翻译模式

|  |  |  |
| --- | --- | --- |
| S | id := E | { S.code := E.code || gen(id .place ‘:=’ E.place) } |
| E | id | { E.place := id .place } |
| E | int | { E.place := newtemp;E.code := gen (E.place ‘:=’ int .val) } |
| E | real | { E.place := newtemp;E.code := gen (E.place ‘:=’ real .val) } |
| E | E1 + E2 | { E.place := newtemp; E.code := E1.code || E2.code || gen (E.place ‘:=’ E1.place ‘+’ E2.place) } |
| E | E1 E2 | { E.place := newtemp; E.code := E1.code || E2.code || gen (E.place ‘:=’ E1.place ‘’E2.place) } |
| E | -E1 | { E.place := newtemp; E.code := E1.code || gen (E.place ‘:=’ ‘uminus’ E1.place) } |
| E | (E1) | { E.place := E1.place ; E.code := E1.code } |

### 2. 说明语句的翻译模式

说明语句的翻译模式如表 6-2所示：

表 6-2 说明语句翻译模式

|  |  |  |
| --- | --- | --- |
| V | V1 ; T | { L.type := T.type; L.offset := V1.width ; L.width := T.width }  L{ V.type := make\_product\_3 (V1.type, T.type, L.num);V.width := V1 .width + L.num  T.width } |
| V | ε | { V.type := <>; V.width := 0 } |
| T | boolean | { T.type := bool ; T.width := 1 } |
| T | integer | { T.type := int ; T.width := 4 } |
| T | real | { T.type := real ; T.width := 8 } |
| T | array [ num ] of T1 | { T.type := array(1.. num.lexval,T1.type) ;T.width := num.lexval  T1.width } |
| T | ^T1 | { T.type := pointer(T1.type) ; T.width := 4 } |
| L |  | { L1. type := L. type ; L1. offset := L. offset ;L1. width := L. width ; }  L1 , id{ enter (id.name, L. type, L. offset + L1.num  L. width) ;L.num := L1.num +1 } |
| L | id | { enter (id.name, L. type, L. offset) ; L.num := 1} |

### 布尔表达式的翻译模式（L-翻译模式）

布尔表达式翻译模式如表 6-3所示：

表 6-3 L-翻译模式

|  |  |  |
| --- | --- | --- |
| E | E1˅E2 | { E1.true := E.true; E1.false := newlabel }  { E2.true := E.true; E2.false := E.false }  { E.code := E1 .code || gen (E1.false ‘:’) || E2 .code } |
| E | E1ΛE2 | { E1.true := newlabel; E1.false := E.false }  { E2.true := E.true; E2.false := E.false } |
| E | !E1 | { E1.true := E.false; E1.false := E.true }  { E.code := E1.code } |
| E | （E1) | { E1.true := E.true; E1.false := E.false }  { E.code := E1.code } |
| E | id1 rop id2 | { E.code :=gen (‘if‘ id1.place rop.op id2.place ‘goto’ E.true ) || gen (‘goto’ E. false) } |
| E | true | { E.code := gen (‘goto’ E.true) } |
| E | false | { E.code := gen (‘goto’ E. false) } |

### 控制语句的翻译模式

控制语句翻译模式如表 6-4所示：

表 6-4 控制语句翻译模式

|  |  |  |
| --- | --- | --- |
| P | D ; | { S.next := newlabel; S.break := S.next }  S{ P.code := S.code || gen(S.next ‘:’) } |
| S | if | { E.true := newlabel; E.false := S.next }  E then{ S1.next := S.next; S1.break := S.break }  S1{ S.code := E.code || gen(E. true ‘:’) || S1.code } |
| S | if | { E.true := newlabel; E.false := newlabel }  E then{ S1.next := S.next; S1.break := S.break }  S1 else{ S2.next := S.next; S2.break := S.break }  S2{ S.code := E.code || gen(E.true ‘:’) || S1.code || gen(‘goto’ S.next) || gen(E.false ‘:’) || S2.code } |
| S | while | { E.true := newlabel; E.false := S.next }  E do{ S1.next := newlabel; S1.break := S.next }  S1{ S.code := gen(S1.next ‘:’) || E.code || gen(E.true ‘:’) || S1.code || gen(‘goto’ S1.next) } |
| S |  | { S1.next := newlabel; S1.break := S.break }  S1 ；{ S2.next := S.next; S2.break := S.break }  S2{ S.code := S1.code || gen(S1.next ‘:’) || S2.code } |
| S | break ; | { S.code := gen(‘goto’ S.break) } |

## 6.2 中间代码生成过程

### 1. 定义opn结构体：

opn结构体中包含类型、种类、层号、偏移量等信息。

struct opn {

int kind; //标识联合成员的属性

int type; //标识操作数的数据类型

union {

int const\_int; //整常数值，立即数

float const\_float; //浮点常数值，立即数

char const\_char[3]; //字符常数值，立即数

char id[33]; //变量或临时变量的别名或标号字符串

};

int level; //变量的层号，0表示是全局变量，数据保存在静态数据区

int offset; //偏移量，目标代码生成时用

};

### 2. 定义codenode结构体：

codenode结构体其采用双向循环链表的方式存储中间代码。

typedef struct codenode { //三地址TAC代码结点,采用单链表存放中间语言代码

int op;

struct opn opn1, opn2, result;

struct codenode\* next, \* prior;

} codenode;

### 定义ASTNode结构体：

对于AST中的结点，需要考虑设置以下属性，在遍历过程中，根据翻译模式给出的计算方法完成属性的计算。

place：记录该结点操作数在符号表中的位置序号，这里包括变量在符号表中的位置，或每次完成了计算后，中间结果需要用一个临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，作用域可以嵌套，不同作用域中的变量可以同名，mini-c语言和C语言一样采用就近优先的原则，但在中间语言中，没有复合语句区分层次，直接根据变量名对变量进行操作，无法区分不同作用域的同名变量，所以每次登记一个变量到符号表中时，会多增加一个别名（alias）的表项，通过别名实现数据的唯一性。翻译时，对变量的操作替换成对别名的操作，别名命名形式为v+序号。生成临时变量时，命名形式为temp+序号，在填符号表时，可以在符号名称这栏填写一个空串，临时变量名直接填写到别名这栏。

type：一个结点表示数据时，记录该数据的类型，用于表达式的计算中。该属性也可用于语句，表示语句语义分析的正确性（OK或ERROR）。

offset：记录外部变量在静态数据区中的偏移量以及局部变量和临时变量在活动记录中的偏移量。另外对函数，利用该数据项保存活动记录的大小。

width：记录一个结点表示的语法单位中，定义的变量和临时单元所需要占用的字节数，借此能方便地计算变量、临时变量在活动记录中偏移量，以及最后计算函数活动记录的大小。

code：记录中间代码序列的起始位置，如采用链表表示中间代码序列，该属性就是一个链表的头指针。

Etrue和Efalse：该结点布尔表达式值为真、假时要转移的程序位置（标号字符串形式）。此属性仅对控制语句中的布尔表达式结点有效，其它情况属性值都是空串。

Snext：该结点的语句序列执行完后，要转移到的程序位置（标号字符串形式）。

struct ASTNode {

//以下对结点属性定义没有考虑存储效率，只是简单地列出要用到的一些属性

//int kind;

enum node\_kind kind; //节点类型

union {

char type\_id[33]; //由标识符生成的叶结点

int type\_int; //由整常数生成的叶结点

float type\_float; //由浮点常数生成的叶结点

char type\_char[3]; //由字符类型生成的叶节点

};

struct ASTNode\* ptr[4]; //由kind确定有多少棵子树

int place; //存放（临时）变量在符号表的位置序号

char Etrue[15], Efalse[15]; //对布尔表达式的翻译时，真假转移目标的标号

char Snext[15]; //结点对应语句S执行后的下一条语句位置标号

struct codenode\* code; //该结点中间代码链表头指针

int type; //用以标识表达式结点的类型

int pos; //语法单位所在位置行号

int offset; //偏移量

int width; //占数据字节数

int num; //计数器，可以用来统计形参个数

};

### 新增的函数

（1）newAlias函数：生成一个新的别名

char \*newAlias()

{

static int no = 1;

char s[10];

//itoa(no++, s, 10);

snprintf(s, 10, "%d", no++);

return strcat0("v", s);

}

（2）newLabel函数：生成一个新的标号。

char \*newLabel()

{

static int no = 1;

char s[10];

//itoa(no++, s, 10);

snprintf(s, 10, "%d", no++);

return strcat0("label", s);

}

（3）newTemp函数：生成一个新的临时变量。

char \*newTemp()

{

static int no = 1;

char s[10];

// itoa(no++, s, 10);

snprintf(s, 10, "%d", no++);

return strcat0("temp", s);

}

（4）genIR函数：生成一条TAC代码的结点组成的双向循环链表，返回头指针。

struct codenode \*genIR(int op, struct opn opn1, struct opn opn2, struct opn result)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = op;

h->opn1 = opn1;

h->opn2 = opn2;

h->result = result;

h->next = h->prior = h;

return h;

}

（5）genLable函数：生成一条标号语句，返回头指针。

struct codenode \*genLabel(char \*label)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = LABEL;

strcpy(h->result.id, label);

h->next = h->prior = h;

return h;

}

（6）genGoto函数：生成GOTO语句，返回头指针。

struct codenode \*genGoto(char \*label)

{

struct codenode \*h = (struct codenode \*)malloc(sizeof(struct codenode));

h->op = GOTO;

strcpy(h->result.id, label);

h->next = h->prior = h;

return h;

}

1. merge函数：合并双向循环链表。

struct codenode \*merge(int num, ...)

{

struct codenode \*h1, \*h2, \*p, \*t1, \*t2;

va\_list ap;

va\_start(ap, num);

h1 = va\_arg(ap, struct codenode \*);

while (--num > 0)

{

h2 = va\_arg(ap, struct codenode \*);

if (h1 == NULL)

h1 = h2;

else if (h2)

{

t1 = h1->prior;

t2 = h2->prior;

t1->next = h2;

t2->next = h1;

h1->prior = t2;

h2->prior = t1;

}

}

va\_end(ap);

return h1;

}

1. prnIR函数：输出中间代码。

void prnIR(struct codenode \*head)

{

char opnstr1[32], opnstr2[32], resultstr[32];

struct codenode \*h = head;

FILE \*file = fopen("code.txt", "w+");

do

{

if (h->opn1.kind == INT)

sprintf(opnstr1, "%d", h->opn1.const\_int);

if (h->opn1.kind == FLOAT)

sprintf(opnstr1, "%f", h->opn1.const\_float);

if (h->opn1.kind == ID)

sprintf(opnstr1, "%s", h->opn1.id);

if (h->opn1.kind == CHAR)

sprintf(opnstr1, "%s", h->opn1.const\_char);

if (h->opn2.kind == INT)

sprintf(opnstr2, "%d", h->opn2.const\_int);

if (h->opn2.kind == FLOAT)

sprintf(opnstr2, "%f", h->opn2.const\_float);

if (h->opn2.kind == ID)

sprintf(opnstr2, "%s", h->opn2.id);

/\*if (h->opn2.kind = CHAR)

sprintf(opnstr2, "#%s", h->opn2.const\_char);\*/

sprintf(resultstr, "%s", h->result.id);

switch (h->op)

{

case ASSIGNOP:

#if show

printf("%s := %s\n", resultstr, opnstr1);

#endif

fprintf(file, "%s = %s\n", resultstr, opnstr1);

break;

case PLUS:

case MINUS:

case STAR:

case DIV:

case MOD:

#if show

printf(" %s := %s %c %s\n", resultstr, opnstr1,

h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op == STAR ? '\*' : h->op == DIV ? '/' : '%', opnstr2);

#endif

fprintf(file, "%s = %s %c %s\n", resultstr, opnstr1,

h->op == PLUS ? '+' : h->op == MINUS ? '-' : h->op == STAR ? '\*' : h->op == DIV ? '/' : '%', opnstr2);

break;

case FUNCTION:

#if show

printf("\nFUNCTION %s :\n", h->result.id);

#endif

fprintf(file, "FUNCTION %s\n", h->result.id);

break;

case PARAM:

#if show

printf(" PARAM %s\n", h->result.id);

#endif

fprintf(file, "PARAM %s\n", h->result.id);

break;

case LABEL:

#if show

printf("LABEL %s :\n", h->result.id);

#endif

fprintf(file, "LABEL %s :\n", h->result.id);

break;

case GOTO:

#if show

printf(" GOTO %s\n", h->result.id);

#endif

fprintf(file, "GOTO %s\n", h->result.id);

break;

case JLE:

#if show

printf(" IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

if (h->result.kind == boolean)

fprintf(file, "%s = %s <= %s\n", resultstr, opnstr1, opnstr2);

else

fprintf(file, "IF %s <= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JLT:

#if show

printf(" IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

fprintf(file, "IF %s < %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JGE:

#if show

printf(" IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

fprintf(file, "IF %s >= %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case JGT:

#if show

printf(" IF %s > %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

fprintf(file, "IF %s > %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case EQ:

#if show

printf(" IF %s == %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

fprintf(file, "IF %s == %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case NEQ:

#if show

printf(" IF %s != %s GOTO %s\n", opnstr1, opnstr2, resultstr);

#endif

fprintf(file, "IF %s != %s GOTO %s\n", opnstr1, opnstr2, resultstr);

break;

case ARG:

#if show

printf(" ARG %s\n", h->result.id);

#endif

fprintf(file, "ARG %s\n", h->result.id);

break;

case CALL:

if (!strcmp(opnstr1, "write"))

{

#if show

printf(" CALL %s\n", opnstr1);

#endif

fprintf(file, "CALL %s\n", opnstr1);

}

else

{

#if show

printf(" %s := CALL %s\n", resultstr, opnstr1);

#endif

fprintf(file, "%s = CALL %s\n", resultstr, opnstr1);

}

break;

case RETURN:

if (h->result.kind)

{

#if show

printf(" RETURN %s\n", resultstr);

#endif

fprintf(file, "RETURN %s\n", resultstr);

}

else

{

#if show

printf(" RETURN\n");

#endif

fprintf(file, "RETURN\n");

}

break;

case BLOCK:

#if show

printf("BLOCK %s :\n", h->result.id);

#endif

fprintf(file, "BLOCK %s :\n", h->result.id);

break;

default:

break;

}

h = h->next;

} while (h != head);

fclose(file);

}

## 6.3 中间代码生成结果

通过指令：

gcc -o parser ast.c analysis.c parser.tab.c lex.yy.c blocks.c optimize.c

编译得到parser.exe程序，然后用parser.exe测试test.c得到的结果保留在三个文本文件code.txt、temp.txt、optimize.txt中，分别如图 6-1 中间代码生成结果所示：

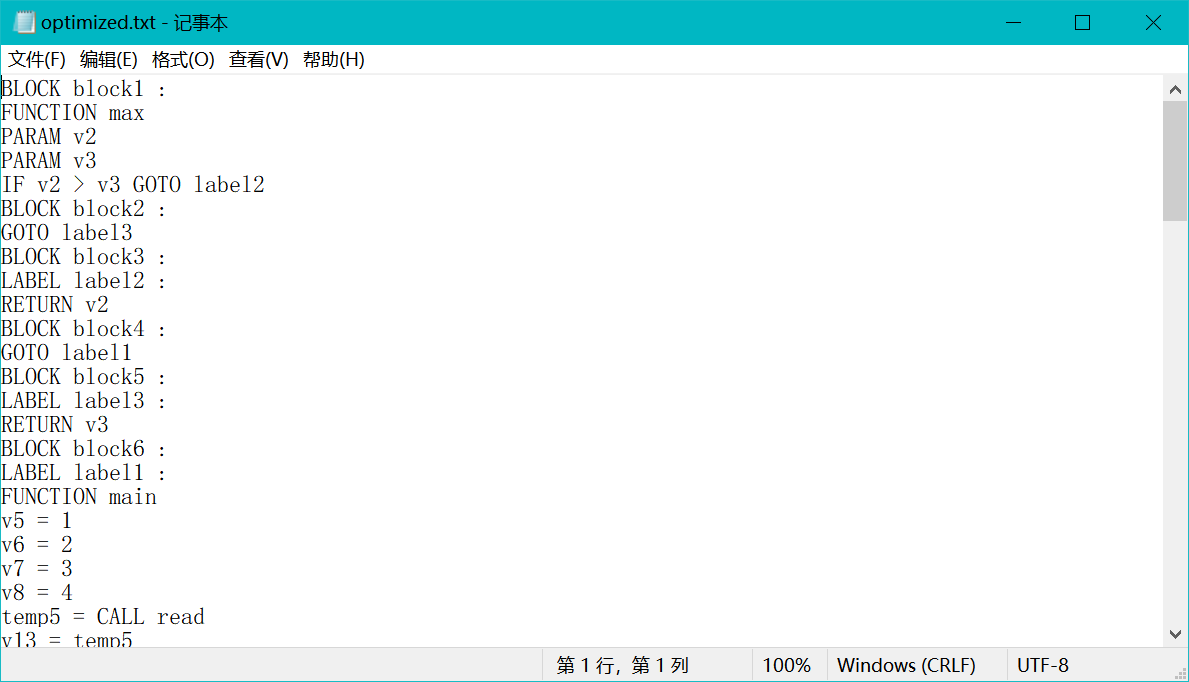
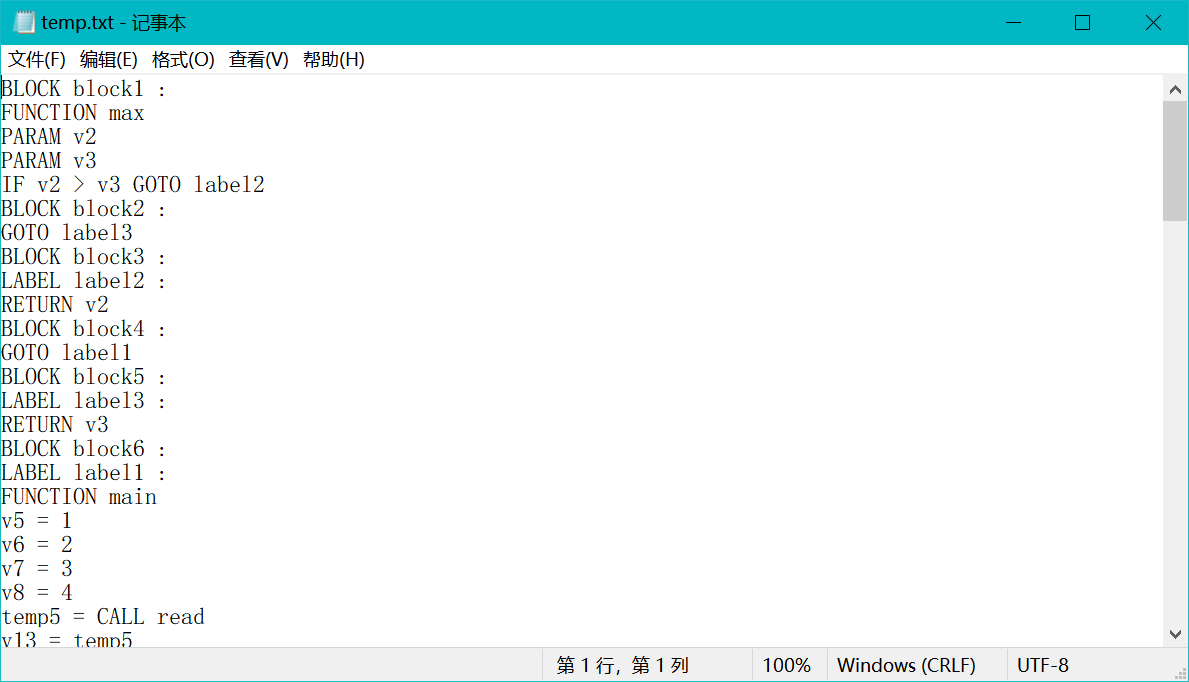
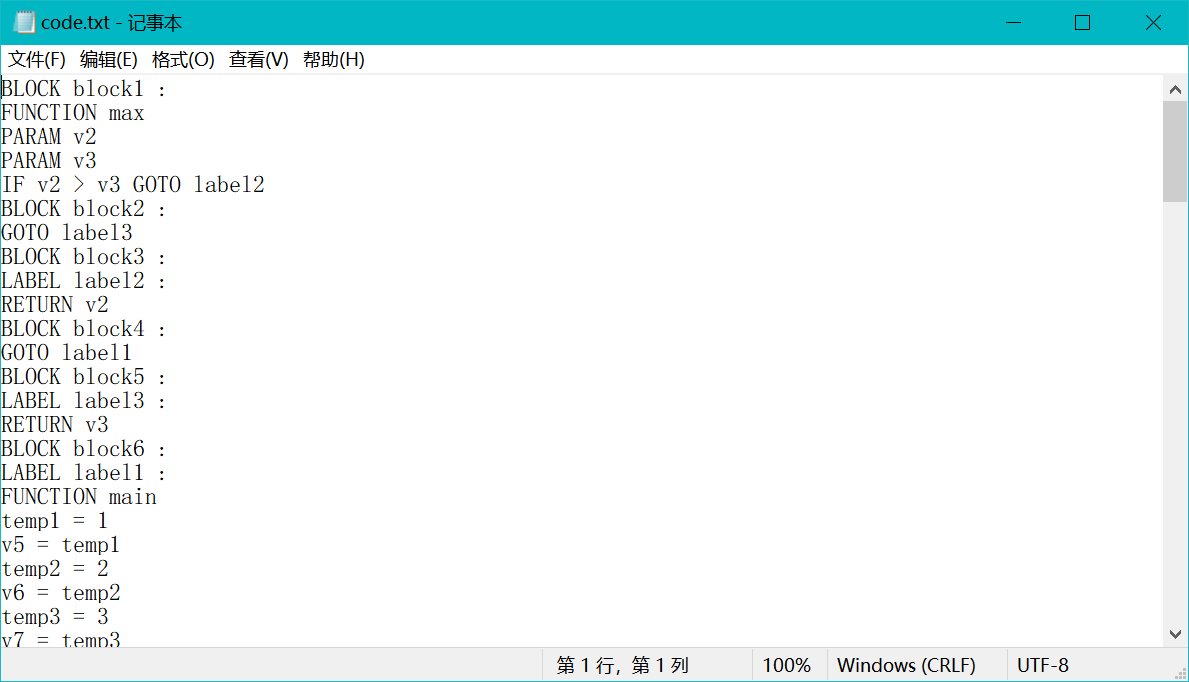


图 6-1 中间代码生成结果

# 7 总结

## 7.1 实验完成情况

整个实验中，完成内容有：

1. 语言的定义；
2. 词法分析器设计与实现；
3. 语法分析器的设计与实现；
4. 符号表的管理和属性计算；
5. 静态语义分析；
6. 中间代码生成。

为完成的内容有：代码优化和目标代码生成。

## 7.2 实验感想

整个实验过程能算得上是整个大学生涯中最复杂的实验了。

首先就是编译原理这门课自身的晦涩程度比较高，很多知识比较抽象，理解起来有些许困难，然后把这些知识运用到实际工作和实验中就更加困难了。

然后就是工作量巨大，由于编译原理实验本身只能用普通的文本编辑器编写，每一行代码的书写都很难通过编译器来找到错误，只能在测试中慢慢寻找；而且后期的一些词法语法分析实验、静态语义分析、中间代码生成这些都需要设计大量的代码，工作量大复杂度高。

但无论怎么说，最终还是顺利完成了实验。虽然前期会被折磨，但在搞懂了编译器的工作原理后，并自己实现了一个建议的编译器后，内心的成就感还是很高的。随着后面实验工作的展开，也对前期的语言定义做了一些相应调整，使得编译器的实验能够尽量简单，不给自己找麻烦；且在词法和语法分析的过程中，最初的代码会有归约冲突的问题，然后花费了大量时间取解决这些问题，避免在后续的实验中产生不必要的麻烦。

实验总体难度偏高，导致后续的实验七和实验八的实现有心无力，最后放弃了这两个实验。不过前面的实验过程都顺利完成，颇有收获。

## 7.3 展望

希望在后续的学习过程中能对编译原理有更加深刻的理解，同时也希望老师能多讲解一些实验的相关内容，而不是只给出一些抽象的指导文档，毕竟看这些文档只是告知了我们需要做什么，而究竟改怎么做和如何判断自己做的好与坏总是十分模糊的。