华中科技大学软件学院

**编译技术实验指导教程**

编译技术课程组

本实验指导教程作为编译技术课程的课程实验和课程设计的参考，对编译器的构造给出了一些参考性的指导意见。本文档不是一个完整使用手册，所以在阅读时，还需要阅读参考文献，并上网查询相关的资料，最后根据自己的理解，选择一种适合自己的技术线路，完成自定义的高级语言编译器的构造。

# 1定义高级语言

编译课程的实验，第一个需要完成的工作，就是要定义一个待实现其编译器的语言，用上下文无关文法定义该语言，并给该语言起一个有意义的名称。后续的工作就是完成该语言的编译器。

## 1.1 mini-c语言的文法

本节给出的是一个简化的C语言的文法，不妨将其称为mini-c。在后续的章节中，主要介绍mini-c的编译过程，给出mini-c语言编译程序构造各阶段实现的指导建议，借此理掌握解编译实现的主要技术线路。Mini-c文法如下：

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε // <程序>→<外部声明>|<程序><外部声明>

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt //声名->变量声名|函数声名

Specifier→int | float|char //类型说明符，添加了char类型

ExtDecList→VarDec | VarDec , ExtDecList //全局变量声名，可以同时声名同一类型的多个变量

VarDec→ID |VarDec [ INT ] //基础变量 ID是变量名，添加了数组类型

FucDec→ID ( VarList ) | ID ( ) //函数申明 ID是函数名，参数可以有多个也可以无

VarList→ParamDec , VarList | ParamDec //参数列，可以是一个也可以是多个

ParamDec→Specifier VarDec //参数声名 VarDec是变量名

CompSt→{ DefList StmList } //函数体 ->声名列表 语句列表

StmList→Stmt StmList | ε //语句列表->语句 语句列表

Stmt→Exp ; | CompSt | return Exp ;| if ( Exp ) Stmt | if ( Exp ) Stmt else Stmt | while ( Exp ) Stmt //语句-> if/while

DefList→Def DefList | ε //复合语句的变量定义部分->声名 声名列表

Def→Specifier DecList ; //声名，可以同时声名同一类型的多个变量

DecList→Dec | Dec , DecList //局部变量名列表

Dec→VarDec | VarDec = Exp //局部变量->变量名|变量赋值

Exp →Exp =Exp | Exp && Exp | Exp || Exp | Exp < Exp | Exp <= Exp

| Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp

| Exp + Exp | Exp - Exp | Exp \* Exp | Exp / Exp | ID | INT | FLOAT

| ( Exp ) | - Exp | ! Exp | ID ( Args ) | ID ( )| ID[Exp]// 参数表达式 添加了对数组调用的支持

Args→Exp , Args | Exp

以上只是给出了一个很简单的语言文法，数据类型只支持整型和浮点；函数只有定义，没有原型声明；以及不支持数组，结构等等。

## 1.1语言的扩展

上节定义的mini-c语言，形式过于简单，以至实现一些很基本的程序功能都会比较困难。所以实验时，要求自行进行扩展，重新定义符合实验要求的语言文法。下面就对扩展部分的一些较重要的语法成分进行说明。

1. 数组类型

在高级语言中，对数组类型的支持，应该包括数组名的定义，数组元素的引用（下标变量）这两个方面。

数组名的定义可以和基本变量（对应的语法符号是VarDec）放在一起，例如： int a,b[2][3]。Mini-c中，基本变量的规则是：VarDec→ID，这样只能定义简单变量名，而数组名标识符后需要说明数组的维数和每一维大小,并且数组可以是多维的。这样需要对规则修改成：VarDec→ID | VarDec [ INT ]。对数组名采用递归规则，这样可表示成任意维数的数组名。

数组元素的使用，在不考虑C语言的指针情况下，就是对数组基本元素的访问，等同于同类型的变量。这时需要对表达式的元素进行扩展，由于可以有多个下标，所以也需要采用递归规则，将Exp的规则集中加上：Exp→Exp[Exp]，其中下标部分要用表达式，不能地简单采用整数常量形式，否则数组的使用失去实用价值。

1. 结构类型

结构类型是一个构造性数据类型，在程序设计中有着非常重要的作用。通常有三种形式使用结构类型：第一种是给出结构类型的完整定义，包括结构类型名，结构成员；第二种是采用匿名结构类型，无结构类型名，仅有结构成员；第三种是仅有结构类型名，这时表示前面已经给出了该结构类型名的完整定义，此处可直接使用结构类型名作为类型说明符。

结构类型名和基本类型名都可以用来定义变量等，是Specifier的一种形式，在文法设计时，结构类型名对应一个语法单元StructSpecifier，依据三种形式，合并成一组规则：

StructSpecifier → struct StructName { DefList } | struct ID

StructName → ID | ε

通过结构类型名声明结构变量后，需要注意的是，对结构变量可以忽略按结构变量名进行整体操作地形式，只需要简单地处理对成员地访问，在Exp中增加：Exp → Exp. ID 这条规则，完成对ID表示的成员进行访问。另外，根据这组文法规则，结构体成员名称说明是直接采用变量说明形式，这样就相当于对结构成员可以有个默认初值，这个不符合C语言的定义。对这个问题的解决，可以考虑另外定义一组规则单独描述结构成员，或者把这个问题放在语义处理部分，在定义结构成员时，不允许有初始化成员值的形式。

1. 面向对象的语言，在本实验指导中，不做介绍，具体参考参考文献[3]。

# 2词法分析与语法分析

第一个实验，构建词法语法分析器，既可以按教材中介绍的方法，自行编写程序来完成，也可以使用工具来实现。基于简单和效率方面的考虑，在清楚了词法、语法分析算法原理的基础上，建议通过联合使用2个工具（Flex和Bison）来构造词法、语法分析程序，语法正确后生成抽象语法树。

根据语言的词法规则，按Flex要求的格式，编辑Lex.l文件（这里文件名可以自行定义，但扩展名一定要是.l），使用Flex编译后即可得到词法分析源程序Lex.yy.c，通过调用yylex（）进行词法分析，每当识别出一个单词，将该单词的（单词种类码，单词的自身值）输出或提供给语法分析程序；

根据语言的语法规则，按Bison要求的格式，编辑Parser.y文件（这里文件名可以自行定义，但扩展名一定要是.y），使用Bison编译后即可得到语法分析源程序Parser.tab.c，调用parser（）进行语法分析。

二者联合在一起完成词法与语法分析时，要求统一单词的种类编码，这时可将各个单词在parser.y中逐个以标识符的形式，通过%token罗列出来。在Parser.y文件中，这些标识符在语法规则部分，作为语法规则的终结符；同时用Bison编译后，可生成一个文件Parser.tab.h，该文件中将这些标识符定义为枚举常量，每一个就对应一个（类）单词，这些枚举常量提供给Lex.l使用，每当识别出一类单词时，就可返回对应的种类码（枚举常量）。联合使用FLEX和Bison构造词法、语法分析器的工作流程图示意图如图2-1所示。



**图2-1 使用Flex和Bison构建语法分析器工作流程**

这个流程中，以语法分析作为主体完成语法分析。在语法分析过程中，每当需要读入下一个符号（单词）时，就调用词法分析器，得到一个单词的（单词种类码，单词的自身值），其控制流程如图2-2所示。



**图2-2 词法、语法分析器控制流程**

## 2.1 词法分析

词法分析器的构造技术线路，首选一个就是设计能准确表示各类单词的正则表达式。用正则表达式表示的词法规则等价转化为相应的有穷自动机FA，确定化、最小化，最后依据这个FA编写对应的词法分析程序。

实验中，词法分析器可采用词法生成器自动化生成工具GNU Flex，该工具要求以正则表达式（正规式）的形式给出词法规则，遵循上述技术线路，Flex自动生成给定的词法规则的词法分析程序。于是，设计能准确识别各类单词的正则表达式就是关键。

### 2.1.1 词法分析的任务

高级语言的词法分析器，需要识别的单词有五类：关键字（保留字）、运算符、界符、常量和标识符。依据mini-c语言的定义，在此给出各单词的种类码和相应符号说明：

INT → 整型常量

FLOAT → 浮点型常量

ID → 标识符

ASSIGNOP → =

RELOP → > | >= | < | <= | == | !=

PLUS → +

MINUS → -

STAR → \*

DIV → /

AND → &&

OR → ||

NOT → !

TYPE → int | float

RETURN → return

IF → if

ELSE → else

WHILE → while

SEMI → ；

COMMA → ，

SEMI → ；

LP → （

RP → ）

LC → {

RC → }

这里有关的单词种类码：INT、FLOAT、......、WHILE，每一个对应一个整数值作为其单词的种类码，实现时不需要自己指定这个值，当词法分析程序生成工具Flex和语法分析程序生成器Bison联合使用时，将这些单词符号作为%token的形式在Bison的文件(文件扩展名为.y)中罗列出来，就可生成扩展名为.h的头文件，以枚举常量的形式给这些单词种类码进行自动编号。这些标识符在Flex文件(文件扩展名为.l)中，每个表示一个（或一类）单词的种类码，在Bison文件(文件扩展名为.y)中，每个代表一个终结符。有关具体细节在后面介绍Bison时再进行叙述。

### 2.1.2 FLEX代码结构

本文不是一个工具的使用说明书，只是纲领性地叙述如何使用工具构造词法、语法分析程序，有关Flex的详细使用方法参见文献[1][2]。使用工具Flex生成词法分析程序时，按照其规定的格式，生成一个Flex文件，Flex的文件扩展名为.l的文本文件，假定为lex.l，其格式为：

**定义部分**

**%%**

**规则部分**

**%%**

**用户子程序部分**

这里被%%分隔开的三个部分都是可选的，没有辅助过程时，第2个%%可以省略。

第一个部分为定义部分，其中可以有一个**%{ 到%}**的区间部分，主要包含c语言的一些宏定义，如文件包含、宏名定义，以及一些变量和类型的定义和声明。会直接被复制到词法分析器源程序lex.yy.c中。**%{ 到%}**之外的部分是一些正则式宏名的定义，例如：id [A-Za-z][A-Za-z0-9]\*，定义了一个表示标识符的宏名id，这些宏名在后面的规则部分会用到。

第二个部分为规则部分，一条规则的组成为：

**正则表达式 动作**

这部分以正则表达式的形式，罗列出所有种类的单词，表示词法分析器一旦识别出该**正则表达式**所对应的单词，就执行**动作**所对应的操作，返回单词的种类码。在这里可写代码显示（种类编码，单词的自身值），观察词法分析每次识别出来的单词，仅作为实验检查的依据。

每当词法分析器识别出一个单词后，将该单词对应的字符串保存在yytext中，其长度为yyleng，供后续使用。

第三个部分为用户子程序部分，这部分代码会原封不动的被复制到词法分析器源程序lex.yy.c中。

### 2.1.3 正则表达式形式

**（1）基本语法**

. 匹配除换行符"\n"外的任意单个字符。

[] 匹配方括号中字符的任意一个。例如[0123456789]表示0-9间的任意一个符号；用"-"指示字符的范围，[0-9]也表示0-9间的任意一个符号；如果第一个符号是“^”,则表示对方括号内的字符取补，例如[^0-9a-zA-Z]，表示所有非数字、字母的符号。

\* 闭包操作，匹配前面Flex正则表达式的零次或多次出现。例如{a}\*，表示零个或多个字母a组成的符号串。

+ 正闭包操作，匹配前面Flex正则表达式的1次或多次出现。例如{a}+，表示一个或多个字母a组成的符号串。

? 匹配零个或一个正则表达式，代表出现在它之前的项目有或没有均可。例如：-？[0-9]+,表示一个数字串，前面的负号可有可无。

{} 根据括号内的内容不同而不同。如果是数字，单个数字{n}意味着前面的模式重复n次，如：[A-Z]{3}；{n，}表示在它之前出现的项目至少重复n次；{n,m}表示在它出现之前的项目重复n次到m次。如果在花括号中包含的是一个定义部分定义了的名字，则表示该名字对应的正则表达式。例如在定义部分定义了名字：digit [0-9],则{digit}表示一个数字字符，{digit}{1,3}表示连续的一至三个数字字符。

| 选择操作，匹配前后的任一表达式。例如true | false表示这2个符号串中的任意一个。

() 将一系列Flex正则表达式归为一组，结合\*、+或？使用。例如abc+表示ab后面连续出现一到多个c；(abc)+表示符号串abc连续出现一到多次

"..." 匹配引号内的内容，如："while" 表示符号串while。采用这种形式，非常方便地表示c语言各种关键字的正则表达式。

/ 只有当有后面的表达式跟随时才匹配前面的表达式。例如x/y表示仅当x后面跟着y时才识别x。

^ Flex正则表达式的第一个字符，它匹配行的开始；在方括号中用于否定，其它方面没有特殊情况。

$ Flex正则表达式的最后一个字符，它匹配行的结尾-其他方面没有特殊情况。

<> 位于模式开头的尖括号内的一个或一列使那个模式只应用于指定的起始状态。

<<EOF>> 在Flex中，这个特殊的模式<<EOF>>匹配文件的结尾。

\ 和后面符号合起来表示各种转义序列，很多与C语言类似。

\d  匹配一个数字字符。等价于[0-9]。  
\D  匹配一个非数字字符。等价于[^0-9]。  
\f  匹配一个换页符。等价于\x0c和\cL。  
\n  匹配一个换行符。等价于\x0a和\cJ。  
\r  匹配一个回车符。等价于\x0d和\cM。  
\s  匹配任何空白字符，包括空格、制表符、换页符等等。等价于[\f\n\r\t\v]。

\S  匹配任何非空白字符。等价于[^\f\n\r\t\v]。  
\t  匹配一个制表符。等价于\x09和\cI。  
\v  匹配一个垂直制表符。等价于\x0b和\cK。  
\w  匹配包括下划线的任何单词字符。等价于’[A-Za-z0-9\_]’。  
\W  匹配任何非单词字符。等价于’[^A-Za-z0-9\_]’。

  任何不属于上面形式的字符在正则表达式中仅匹配该字符自身。

**（2）正则式举例**

\d+ //匹配非负整数（正整数+0）  
[0-9]\*[1-9][0-9]\* //匹配正整数  
((-\d+)|(0+)) //匹配非正整数（负整数+0）  
-[0-9]\*[1-9][0-9]\* //匹配负整数  
-?\d+ //匹配整数  
\d+(\.\d+)? //匹配非负浮点数（正浮点数+0）  
(([0-9]+\.[0-9]\*[1-9][0-9]\*)|([0-9]\*[1-9][0-9]\*\.[0-9]+)|([0-9]\*[1-9][0-9]\*)) //匹配正浮点数  
((-\d+(\.\d+)?)|(0+(\.0+)?)) //匹配非正浮点数（负浮点数+0）  
(-(([0-9]+\.[0-9]\*[1-9][0-9]\*)|([0-9]\*[1-9][0-9]\*\.[0-9]+)|([0-9]\*[1-9][0-9]\*)))

//匹配负浮点数  
(-?\d+)(\.\d+)? //匹配浮点数  
 附录1给出了第1章中定义的mini-c语言的部分词法分析程序lex.l，还缺注释（包括行注释和块注释）的处理，实验时需要补全。对该文件使用Flex进行翻译，**命令形式为：flex lex.l**，即可得到词法分析器的c语言源程序文件lex.yy.c。

## 2.2 语法分析

语法分析采用生成器自动化生成工具GNU Bison（前身是YACC），该工具采用了LALR（1）的自底向上的分析技术，完成语法分析。在实验的语法分析阶段，当语法正确时，生成抽象语法树，作为后续语义分析的输入。Bison程序文件的扩展名为.y，附录2中的文件parser.y给出了mini-c语言的语法分析Bison程序。有关Bison的使用方法参见文献[1]、[2]。parser.y的格式为：

**%{**

**声明部分**

**%}**

**辅助定义部分**

**%%**

**规则部分**

**%%**

**用户函数部分**

### 2.2.1 声明部分

%{到%}间的声明部分内容包含语法分析中需要的头文件包含，宏定义和全局变量的定义等，这部分会直接被复制到语法分析的C语言源程序中。

### 2.2.2 辅助定义部分

在该部分，可以处理实验中要用到的几个主要内容：

（1）终结符定义，在Flex和Bison联合使用时，parser.y如何使用lex.l中识别出的单词的种类码？这时需要在parser.y中的%token后面罗列出所有终结符(单词)的种类码标识符，如：

%token ID，INT，IF，ELSE，WHILE

这样就完成了定义终结符ID、INT、IF、ELSE、WHILE。接着可使用命令：**bison -d parser.y** 对语法分析的Bison文件parser.y进行翻译，当使用参数**-d**时，除了会生成语法分析器的c语言源程序文件parser.tab.c外，还会生成一个头文件parser.tab.h，在该头文件中，将所有的这些终结符作为枚举常量，从258开始，顺序编号。这样在lex.l中，使用宏命令 #include “parser.tab.h”，就可以使用这些枚举常量作为终结符（单词）的种类码返回给语法分析程序，语法分析程序接收到这个种类码后，就完成了读取一个单词。

（2）语义值的类型定义，mini-c的文法中，有终结符，例如：ID表示的标识符，INT表示的整常数，IF表示关键字if，WHILE表示关键字while等；同时也有非终结符，如ExtDefList表示外部定义列表，CompSt表示复合语句等。每个符号（终结符和非终结符）都会有一个属性值，这个值的类型默认为整型。实际运用中，属性值的类型会有些差异，如ID的属性值类型是一个字符串，INT的属性值类型是整型。在下一节会介绍，语法分析时，需要建立抽象语法树，这时ExtDefList的属性值类型会是树结点（类型为**struct ASTNode**）的指针。这样各种符号就会对应不同类型，这时可以用联合将这多种类型统一起来：

**%union {**

**int type\_int;**

**float type\_float;**

**char type\_id[32];**

**struct ASTNode \*ptr;**

**. . . . . .**

**}**

将所有符号属性值的类型用联合的方式统一起来后，某个符号的属性值就是该联合中的一个成员的值。

**终结符属性值的类型说明**，需要在%token定义终结符符号时指定其属性对应联合中的哪个成员。例如识别出一个整型常量时，对应的终结符是INT，需要一个整型常数作为自身值，可以使用 %token <type\_int> INT 来说明INT对应联合中的成员type\_int；识别出一个标识符时，对应的终结符是ID，需要一个字符串作为自身值，可以使用 %token <type\_id> ID 来说明ID对应联合中的成员type\_id。这样在parser.y文件中使用文法规则时，直接通过INT或ID，就可以使用整型常数自身值整数或标识符自身值字符串。

**非终结符属性值的类型说明**，对于非终结符，如果需要完成语义计算时，会涉及到非终结符的属性值类型，这个类型对应联合的某个成员，可使用格式：%type <union的成员名> 非终结符。例如parser.y中的：

%type <ptr> program ExtDefList

这表示非终结符program 和ExtDefList属性值的类型对应联合中成员ptr的类型，在本实验中对应一个树结点的指针。

有关终结符和非终结符属性的处理在2.3中会进一步说明。

（3）优先级与结合性定义。对Bison文件进行翻译，得到语法分析分析程序的源程序时，通常会出现报错，大部分是移进和归约(shift/reduce)，归约和归约(reduce /reduce)的冲突类的错误。为了改正这些错误，需要了解到底什么地方发生错误。这时，需要在翻译命令中，加上一个参数**-v**，即命令为： ：**bison -d -v parser.y** 这时，会生成一个文件parser.output。打开该文件，开始几行说明（LALR（1）分析法）哪几个状态有多少个冲突项，再根据这个说明中的状态序号去查看对应的状态进行分析、解决错误，常见的错误一般都能通过单词优先级和结合性的设定解决，例如对表达式Exp，其部分文法规则有：

Exp →Exp + Exp | Exp - Exp | Exp \* Exp | Exp \* Exp

在文法介绍时，明确过该文法是二义性的，这样对于句子a+b\*c，到了符号\*时，可能的操作一个是移进\*，一个是对前面的a+b所对应的Exp+Exp进行归约。同样，对于句子a+b+c，读到第二个+号时，是移进，还是把前面的归约？

这样对文件parser.y进行翻译时，会出现移进和归约的冲突，在parser.output文件中，其对应的某个状态会出现说明：

'+' shift, and go to state 16

'-' shift, and go to state 17

'\*' shift, and go to state 18

'/' shift, and go to state 18

'+' [reduce using rule 12 (exp)]

'-' [reduce using rule 12 (exp)]

'\*' [reduce using rule 12 (exp)]

'/' [reduce using rule 12 (exp)]

前面4条表示遇到这些符号要做的操作是移进，后面4条表示遇到这些符号要做的操作是归约，所以产生冲突。这时的解决方法就是通过设定优先级和结合性来实现：

%left '+' '-'

%left '\*' '/'

left表示左结合，right表示右结合，前面符号的优先级低，后面的优先级高。

另外就是对： Exp →-Exp 单目-的运算优先级高于\*与/，而词法分析时，无论是单目-还是双目-，识别出的种类码都是MINUS，为此，需要在定义一个优先级高的单目-符号UMINUS：

%left '+' '-'

%left '\*' '/'

%right UMINUS

相应对其规则处理为：

Exp →-Exp %prec UMINUS

表示这条规则的优先级等同于UMINUS，高于乘除，这样对于句子-a\*b就会先完成-a归约成Exp，即先处理单目-，后处理\*。

最后就是条件语句的嵌套时的二义性问题的解决发生，参见参考文献[2]中的解决方法。最终要求用Bison对parser.y进行翻译时，一定要去掉此类的全部冲突，避免为后续的工作留下隐患。

### 2.2.3 规则部分

使用Bison采用的是LR分析法，需要在每条规则后给出相应的语义动作,例如对规则：Exp →Exp =Exp，在parser.y中为：

Exp: Exp ASSIGNOP Exp {$$=mknode(2,ASSIGNOP,yylineno,$1,$3); }

规则后面{}中的是当完成归约时要执行的语义动作。规则左部的Exp的属性值用$$表示，右部有2个Exp，位置序号分别是1和3，其属性值分别用$1和$3表示。在附录4中，定义了mknode函数，完成建立一个树结点，这里的语义动作是将建立的结点的指针返回赋值给规则左部Exp的属性值，表示完成此次归约后，生成了一棵子树，子树的根结点指针为$$，根结点类型是ASSIGNOP，表示是一个赋值表达式。该子树有2棵子树，第一棵是$1表示的左值表达式的子树，在mini-c中，简单化为只要求ID表示的变量作为左值，注意实验时，要根据定义的文法做出相应的处理，比如数组下标变量也可以作为左值；第二棵对应是$3的表示的右值表达式的子树，另外yylineno表示赋值语句的行号。

通过使用上述形式，给出所有规则的语义动作，当一个程序使用LR分析法完成语法分析后，如果正确则可生成一棵抽象语法树，抽象语法树在2.4详细叙述。

## 2.3 报错与容错

在使用FLEX进行词法分析、Bison进行语法分析的过程中，要求具有**报错和容错**功能。一旦有词法、语法错误，需要准确、及时地进行报错，给出错误位置以及错误性质。

为了方便标识错误位置，需要记录分析过程中的当前行号。在FLEX中定义了一个内部变量yylineno，当在FLEX文件的定义部分加上%option yylineno后，就可以直接使用这个内部变量了，并且不需要去维护yylineno的值，在词法分析过程中，每次遇到一个回车，yylineno会自动加一。同时在BISON文件的声明部分加上extern int yylineno，就可以共用yylineno的值。

词法分析过程中，一旦遇到识别不了的单词，需要进行报错。处理方法很简单，在FLEX文件的规则部分，前面是能识别出来的所有单词的正则式，最后一个条就是一个符号”.”表示地正则式，表示不能识别出的单词形式，这时结合变量yylineno给出错误信息即可。

语法报错由BISON文件中的yyerror函数负责完成，需要补充的就是错误定位，在源程序的哪一行有错。为了更准确的给出错误性质，可在BISON文件的辅助定义部分加上%error-verbose。

有时希望更准确的标识错误的位置，除了行号以外，还需要标识列号。这时可以利用一个表示位置的类型YYLTYPE，BISON中的每一个语法单元（终结符和非终结符）对应一个YYLTYPE类型的位置信息，YYLTYPE定义形式为：

typedef struct {

int first\_line;

int first\_column;

int last\_line;

int last\_column;

} YYLTYPE;

其中first\_line和first\_column表示该语法单元第一个单词出现的行号和列号，last\_line和last\_column表示该语法单元最后一个单词出现的行号和列号。BISON文件中规则部分可以通过@$、@1、@2等形式引用一条规则各语法单位的位置信息。然而在BISON中并没有主动维护位置信息，导致引用的位置信息错误。为了能正确引用位置信息，需要使用FLEX的内置变量yylloc，yylloc表示当前词法单元所在的位置信息。首先需要在FLEX文件的声明部分加上：

int yycolumn=1;

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

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

yycolumn+=yyleng;

在FLEX文件规则部分的正则式’\n’后面将yycolumn赋值为1，表示一个新的行，列数从1开始。同时在BISON文件的辅助定义部分加上%locations，这样在BISON中，就可以使用yylloc.first\_line和yylloc.first\_column表示当前单词位置的行号和列号，准确地标识错误的位置。同时每条规则后的动作部分通过@$、@1、@2等形式引用该规则各语法单位的位置信息。

编译过程中，待编译的mini-c源程序可能会有多个错误，这时，需要有容错的功能，跳过错误的代码段，继续向后进行语法分析，而不是遇到一个错误就停下来，这个步骤称为**同步**。一次尽可能多地报出源程序的语法错误，减少交互次数，提高编译效率。这时可通过跳过一段源程序代码段到指定的符号，再接着进行语法分析。例如：

Stm →error SEMI

表示对语句进行语法分析时，一旦有错，跳过分号（SEMI），继续向后进行语法分析。可在parser.y中多处设置这种同步操作，比如对外部定义，可再加上：

ExtDef→error SEMI

表示对外部定义进行语法分析时，一旦有错，跳过分号（SEMI），继续向后进行语法分析。多处设置这种同步操作可能会导致使用bison对parser.y进行翻译时出现移进/规约的冲突，但不会影响到对mini-c源程序的正常语法分析，所以对这类同步引起的移进/规约冲突可以不理会。注意，通过同步可能会跳过大量的源程序代码，被跳过的代码中可能还含有其它的语法错误，为了避免大量错误的堆积，可以限制一下同步的次数，到达这个次数时就终止语法分析。

有关原理性的解释和一般使用方法，参见参考文献[1]和文献[2]。

## 2.4 抽象语法树（AST）

在语法分析阶段，一个很重要任务就是生成待编译程序的抽象语法树AST，AST不同于推导树，它去掉了一些修饰性的单词部分，简明扼要地把程序的语法结构表示出来，后续的语义分析、中间代码生成都可以通过遍历抽象语法树来完成。

例如对语句： while (a>=1) a=a-1;

推导树和抽象语法树分别如图2-3的左右2棵树所示。

**图2-3 推导树和抽象语法树的形式**

其中，根据处理方式不同，定义的AST形式可能会存在一些差异。从形式上看，AST比推导树简洁很多，这样后续处理时就方便很多，且效率也会提高。所以语法分析过程中，尽量不要考虑生成推导树的形式。

为了创建AST，需要对文法的各个符号规定一些属性值，如表2-1所示列出了终结符绑定词法分析得到的值，非终结符绑定AST中对应的树结点指针。

**表2-1 文法符号对应属性**

|  |  |
| --- | --- |
| **符 号** | **属 性** |
| ID | 标识符的字符串 |
| INT | 整常数数字 |
| FLOAT | 浮点常数数字 |
| 所有非终结符 | 抽象语法树的结点指针 |
| 其它终结符 | 根据具体文法定义处理 |

由表2-1可见，不同的符号绑定的属性值的类型不一定相同。例如，词法分析器识别出一个正常数123，返回的单词种类码INT，同时INT对应的终结符要对应一个单词自身值(整数123)。

为了将要用到的各种非终结符和终结符的类型统一在一个类型下，如2.2.2中已叙述可采用联合（共用体）这个类型。例如在lex.l和parser.y中，同时定义：

typedef union {

int type\_int;

int type\_float;

char type\_id[32];

struct node \*ptr;

......

} YYLVAL;

这样所有符号属性值的类型就是联合类型YYLVAL了。如何实现不同符号绑定不同的成员哪？ 对终结符，可采用：%token <type\_int> INT，表示INT的属性值对应联合的成员type\_int。例如在lex.l的词法分析中，识别到整常数后，在返回给语法分析器一个整常数的种类码INT的同时，通过yylval.type\_int=atoi(yytext);将整常数的值保存在yylval的成员type\_in中，这里yylval是一个Flex和Bison共用的内部变量，类型为YYLVAL，按这样的方式，在Flex代码中通过yylval的成员保存单词属性值，在Bison代码中就可以通过yylval的成员取出属性值，实现了数据的传递。由于已经建立了绑定关系，语法分析的规则部分的语义处理时，通过终结符INT绑定的属性值可直接取到这个常数值。比如对规则：Exp →INT， 由于终结符INT是规则右部的第一个符号，就可通过$1简单方便地取到识别出的整常数值，不必采用yylval.type\_int的形式提取这个整常数值。

同样可采用：%token <type\_id> ID，表示识别出来一个标识符后，标识符的字符串串值保存在联合成员type\_id中。对非终结符，如果采用%type <ptr> Exp，表示Exp对应绑定成员ptr，即Exp的属性值是一个结点的指针。

在parser.y中处理AST时，所有结点的类型是统一的，所以为区分结点的属性，在生成结点时，要有一个属性kind，用以标识结点类型，明确结点中存放了哪些有意义的信息。结点定义参考附录3中node的定义。

**AST的逻辑结构就是一棵多叉树，实现时需要考虑采用哪种物理结构，这个就需要灵活地运用数据结构课程的知识，可采用：（1）孩子表示法，本指导书中，基于简明以及让读者理解方便的原则，采用的就是结点大小固定的孩子表示法，每个结点有4个指针域，可指向4棵子树。由结点类型kind确定有多少棵子树，显然这会有很多空指针域。如果已经掌握了C++，利用类的封装，继承与多态来定义结点会更好。（2）孩子兄弟表示法（二叉链表），这种方法存储效率要高一些，实现时要清楚结点之间的关系的转换。**

对源程序进行语法分析时，在完成归约的过程中，完成抽象语法树的构造。例如处理INT归约成Exp时，对应规则及语义动作为：

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

需要调用函数mknode生成一个类型为INT的叶结点（度为0），指针赋值给Exp的属性$$，同时将INT的属性值$1（一个整常数）写到结点中type\_int成员域保存。这里yylineno表示语法单元所在的行号。

当处理将Exp1+Exp2归约成Exp时，对应规则及语义动作为：

Exp ： Exp1 PLUS Exp2  {$$=mknode(2,PLUS,yylineno,$1,$3); }

需要调用函数mknode生成一个类型为PLUS、度为2的非叶子结点，结点指针赋值给Exp的属性$$，将Exp1表示的树$1作为Exp的第一棵子树，将Exp2表示的树$3作为Exp的第二棵子树。

如果没有语法错误，最后归约到了文法开始符号，这样就可以获得抽象语法树的根结点指针。再调用display以缩进编排的格式进行显示AST。AST的遍历采用树的先根遍历，有关代码部分参见附录4。

## 2.5 Flex与Bison的安装

下载flex和bison。网址：http://gnuwin32.sourceforge.net/packages/flex.htm和http://gnuwin32.sourceforge.net/packages/bison.htm。仅需下载setup文件进行安装。安装时，设定路径最好不要是在Program Files文件夹里面，因为文件夹名字带空格可能会影响以后的使用。可安装在c:\gnuwin32下面。

其次由于我们使用的flex和bison都是GNU的工具，所以为了方便，采用的C/C++编译器也采用GNU的编译器GCC，当然我们需要的也是Windows版本的GCC了。目前Windows平台 的GCC主要是MinGW编译器，可以到 MinGW的主页http://sourceforge.net/projects/mingw/files/latest/download?source=files下载安装。安装完毕后，将文件夹 c:\gnuwin32\lib 里面的 libfl.a 和liby.a复制到文件夹C:\MinGW\lib里面。接着设置环境变量。右键点击“计算机”，“属性”、“高级系统设置”、“环境变量”，在下面系统变量里面找到PATH，在前面加上2个文件夹的路径： c:\gnuwin32\bin和C:\MinGW\bin。

也可以使用其它的C编译，建议使用codeblocks，可到<http://www.codeblocks.org/>下载。对系统环境变量path的设置还可以使用控制台命令进行临时配置(假定使用codeblocks)：

path c:\gnuwin32\bin; C:\Program Files (x86)\CodeBlocks\MinGW\bin;%path%

我们可以开始两个简单的文件来测试一下。

（1）新建文本文件，更改名称为lex.l，敲入下面代码

%{

int yywrap(void);

%}

%%

%%

int yywrap(void)

{

return 1;

}

（2）新建文本文件，更改名称为yacc.y,敲入下面代码

%{

void yyerror(const char \*s);

%}

%%

program:

;

%%

void yyerror(const char \*s)

{

}

int main()

{

yyparse();

return 0;

}

我们暂且不讨论上面代码的意思。打开控制台，进入到刚才所建立文件（lex.l,yacc.y）所在的文件夹。

1.输入 flex lex.l

2.输入 bison yacc.y

如果我们看到当前文件夹上多了两个文件（yacc.tab.c，lex.yy.c），那么说明lex&&yacc已经安装配置成功。

## 2.6语法分析器的构造

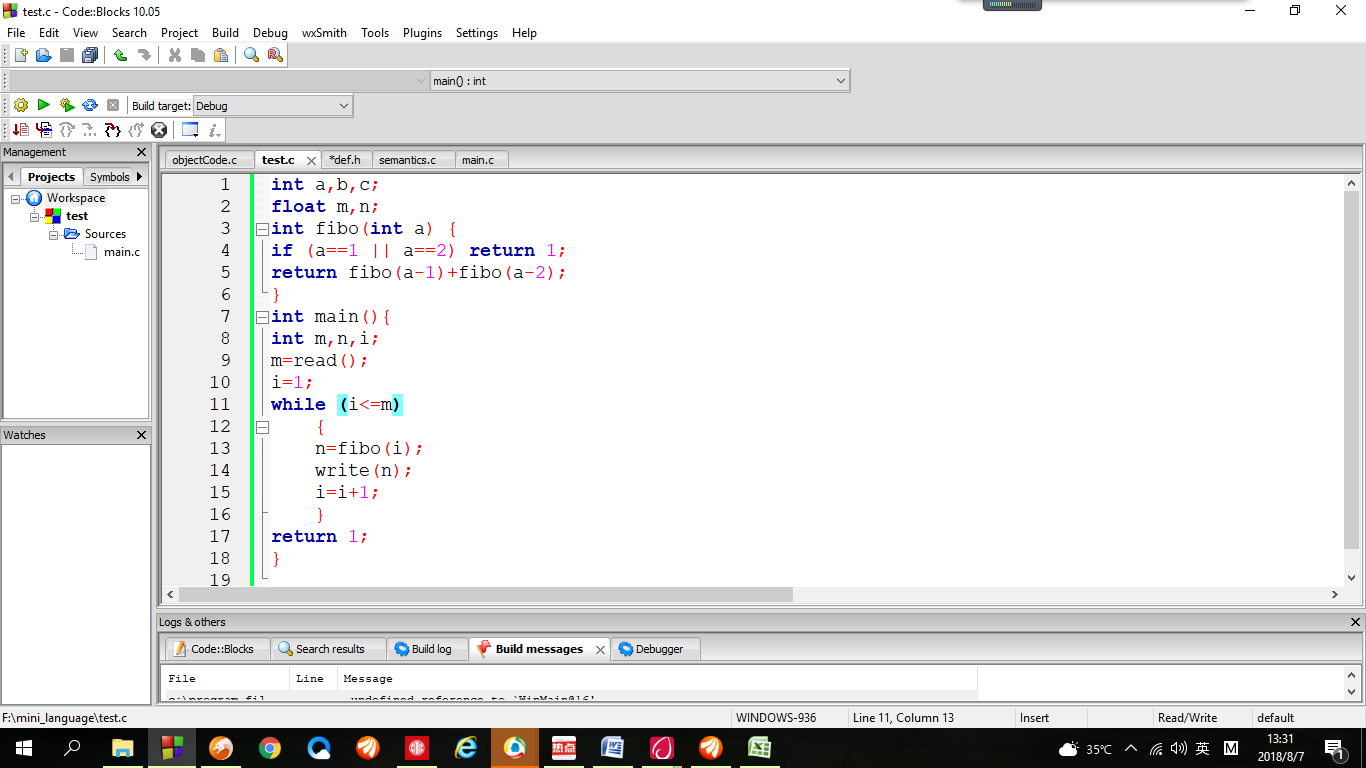
当安装好Flex和Bison后，建立附录1到附录4的文件，并在控制台下执行如下命令序列：

**flex lex.l**  生成lex.yy.c

**bison -d parser.y** 生成parser.tab.c和 parser.tab.h

**gcc -o parser lex.yy.c parser.tab.c ast.c ast.c**  生成 parser.exe

这样就完成了语法分析器**parser**的构造。接着创建如图2-4所示的mini-c程序test.c作为测试程序。



**图2-4 AST生成测试程序**

使用命令：

**parser test.c**

即可完成对文件test.c的词法分析、语法分析，如果出现词法、语法错误，就输出错误位置和错误性质。当词法、语法正确时，显示test.c的抽象语法树如下所示。

**外部变量定义：**

**类型： int**

**变量名：**

**ID： a**

**ID： b**

**ID： c**

**外部变量定义：**

**类型： float**

**变量名：**

**ID： m**

**ID： n**

**函数定义：**

**类型： int**

**函数名：fibo**

**函数形参：**

**类型：int, 参数名：a**

**复合语句：**

**复合语句的变量定义：**

**复合语句的语句部分：**

**条件语句(IF\_THEN)：**

**条件：**

**OR**

**==**

**ID： a**

**INT：1**

**==**

**ID： a**

**INT：2**

**IF子句：**

**返回语句：**

**INT：1**

**返回语句：**

**PLUS**

**函数调用：**

**函数名：fibo**

**第1个实际参数表达式：**

**MINUS**

**ID： a**

**INT：1**

**函数调用：**

**函数名：fibo**

**第1个实际参数表达式：**

**MINUS**

**ID： a**

**INT：2**

**函数定义：**

**类型： int**

**函数名：main**

**无参函数**

**复合语句：**

**复合语句的变量定义：**

**LOCAL VAR\_NAME：**

**类型： int**

**VAR\_NAME：**

**m**

**n**

**i**

**复合语句的语句部分：**

**表达式语句：**

**ASSIGNOP**

**ID： m**

**函数调用：**

**函数名：read**

**表达式语句：**

**ASSIGNOP**

**ID： i**

**INT：1**

**循环语句：**

**循环条件：**

**<=**

**ID： i**

**ID： m**

**循环体：**

**复合语句：**

**复合语句的变量定义：**

**复合语句的语句部分：**

**表达式语句：**

**ASSIGNOP**

**ID： n**

**函数调用：**

**函数名：fibo**

**第1个实际参数表达式：**

**ID： i**

**表达式语句：**

**函数调用：**

**函数名：write**

**第1个实际参数表达式：**

**ID： n**

**表达式语句：**

**ASSIGNOP**

**ID： i**

**PLUS**

**ID： i**

**INT：1**

**返回语句：**

**INT：1**

可以看到，AST重点突出了程序的语法结构，去掉了诸如括号，分号和分隔符等修饰性的单词信息，通过这个AST的显示结果，能很方便地还原出图2-4所示程序代码。

对AST的遍历并显示出来，能帮助我们分析验证语法分析的结果是否正确，同时熟悉使用遍历算法访问结点的次序，这样在后序的语义分析、中间代码的处理过程中，就能非常方便地使用遍历流程完成其对应的编译阶段工作，同时也能给我们在调试程序中提供方便。

# 3语义分析

在前面的实验中，完成了AST的建立，下面就需要对AST进行遍历2次，一次完成本章的语义分析，一次完成下章介绍的中间代码生成。

语义分析阶段，需要借助于符号表以及一些相关的数据结构，完成静态语义检查，可以根据实验定义的语言，检查出如下所述类型的静态语义错误：

（1）使用未定义的变量；

（2）调用未定义或未声明的函数；

（3）在同一作用域，名称的重复定义（如变量名、函数名、结构类型名以及结构体成员名等）。为更清楚说明语义错误，这里也可以拆分成几种类型的错误，如变量重复定义、函数重复定义、结构体成员名重复等；

（4）对非函数名采用函数调用形式；

（5）对函数名采用非函数调用形式访问；

（6）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；

（7）函数调用时实参和形参类型不匹配；

（8）对非数组变量采用下标变量的形式访问；

（9）数组变量的下标不是整型表达式；

（10）对非结构变量采用成员选择运算符“.”；

（11）结构成员不存在；

（12）赋值号左边不是左值表达式；

（13）对非左值表达式进行自增、自减运算；

（14）对结构体变量进行自增、自减运算；

（15）类型不匹配。如数组名与结构变量名间的运算，需要指出类型不匹配错误；有些需要根据定义的语言的语义自行进行界定，比如：32+'A'，10\*12.3，如果使用强类型规则，则需要报错，如果按C语言的弱类型规则，则是允许这类运算的，但需要在后续阶段需要进行类型转换，类型统一后再进行对应运算；

（16）函数返回值类型与函数定义的返回值类型不匹配；

（17）函数没有返回语句（当函数返回值类型不是void时）；

（18）break语句不在循环语句或switch语句中；

（19）continue语句不在循环语句中；

## 3.1 符号表的管理

语义分析这部分的一个非常重要的工作就是符号表的管理，在编译过程中，编译器使用符号表来记录源程序中各种名字的特性信息。所谓“名字”包括：程序名、过程名、函数名、用户定义类型名、变量名、常量名、枚举值名、标号名等，所谓“特性信息”包括： 上述名字的种类、 具体类型、维数（如果语言支持数组）、函数参数个数、常量数值及目标地址（存储单元偏移地址）等。

符号表可以采用多种数据结构实现，实验中可采用不同的数据结构来实现：

（1) **顺序表。本实验指导采用这种方式管理符号表**。此时的符号表symbolTable是一个顺序栈，栈顶指针index初始值为0，每次填写符号时，将新的符号填写到栈顶位置，再栈顶指针加1。

本实验中，为了方便测试程序、观察运行结果，事先默认了2个函数read和write，程序中可以直接使用，所以需要首先将其登记到符号表中。

（2）哈希表。参见文献[3]的P198。

（3）十字链表 。参见文献[2]的P57-58。

（4）多表结构，即每进入一个作用域就创建一张表，每出一个作用域就释放一张表。参见文献[3]的P304-305。

符号表上的操作包括创建符号表、插入表项、查询表项、修改表项、删除表项、释放符号表空间等等。

## 3.2 静态语义分析

语义分析这部分完成的是静态语义分析，主要包括：

（1）控制流检查。控制流语句必须使得程序跳转到合法的地方。例如一个跳转语句会使控制转移到一个由标号指明的后续语句。如果标号没有对应到语句，那么就出现一个语义错误。再者，break、continue语句必须出现在循环语句当中。

在mini-c中没有定义各种转移语句，实验时可以考虑增加上去；教材中有break语句的介绍，可供参考。

（2）唯一性检查。对于某些不能重复定义的对象或者元素，如同一作用域的标识符不能同名，需要在语义分析阶段检测出来。

（3）名字的上下文相关性检查。名字的出现在遵循作用域与可见性的前提下应该满足一定的上下文的相关性。如变量在使用前必须经过声明，如果是面向对象的语言，在外部不能访问私有变量等等。

（4）类型检查包括检查函数参数传递过程中形参与实参类型是否匹配、是否进行自动类型转换等等。

## 3.3 语义程序架构

前面的实验中，完成了AST的生成，现在的工作就是通过遍历AST完成符号表的管理和静态语义分析。

### 3.3.1 AST遍历

AST的遍历采用的是先根遍历，在遍历过程中，访问到了说明部分的结点时，在符号表中添加新的内容；访问到执行语句部分时，根据访问的变量（或函数）名称查询符号表，并分析其静态语义的正确性。

先根遍历AST算法的框架很简单，采用递归算法实现，设T为根结点指针。

（1）如果T为空，遍历结束返回

（2）根据T->kind，即结点类型，可知道该结点有多少棵子树，依次递归访问各子树。

在语义分析阶段，通过遍历访问结点完成各种属性的计算，例如对于一个局部变量说明语句：int a，b；其对应的AST如图3-1所示。



**图3-1 说明语句的AST形式**

当第一次访问到VAR\_DEF结点时，按遍历次序，接着访问TYPE结点，确定TYPE结点的数据类型为INT，回到VAR\_DEF后，该说明语句中的变量列表中各变量的类型确定了，可将此类型属性向下传给结点DEC\_LIST1。接着类型由DEC\_LIST1传到a这个ID结点，这时就明确了a是一个整型变量，查符号表，如果在当前作用域（根据层号）没有定义，就根据a填写一个整型的变量a到符号表中，否则报错，变量重复定义。再接着数据类型INT由DEC\_LIST1传到DEC\_LIST2结点，直到整型变量b完成查表和填写到符号表中。

上述例子的属性计算仅考虑语义分析这部分的需求，但在整个编译过程中，需要同时完成的属性计算还很多，比如访问VAR\_DEF结点时，首先到此结点，由前面的计算结果，已经得到这个说明语句的变量在活动记录中的地址偏移量（offset），这时访问过TYPE结点后，得到该类型变量的宽度值（width），这样a的地址偏移量就为offset，b的地址偏移量为offset+width；最后回到VAR\_TYPE结点时，其说明语句中变量的总宽度计算出为2\*width。所以再遇到VAR\_DEF之后的其它变量说明的结点时，地址偏移量为offset+2\*width。由此给计算出一个函数中所有变量在活动记录中的地址偏移量。在遍历过程中，会涉及到较多的属性计算，需要分清楚哪些是在语义分析中必须的，哪些是后续中间代码生成需要的，语义分析只用做语义分析的事，避免重复计算属性。

### 3.3.2 作用域与符号表操作

在语义分析过程中，各个变量名有其对应的作用域，一个作用域内不允许名字重复，为此，通过一个全局变量LEV来管理，LEV的初始值为0。这样在处理外部变量名，以及函数名时，对应符号的层号值都是0；处理函数形式参数时，固定形参名在填写符号表时，层号为1。由于mini\_C中允许有复合语句，复合语句中可定义局部变量，函数体本身也是一个复合语句，这样在AST的遍历中，通过LEV的修改来管理不同的作用域。

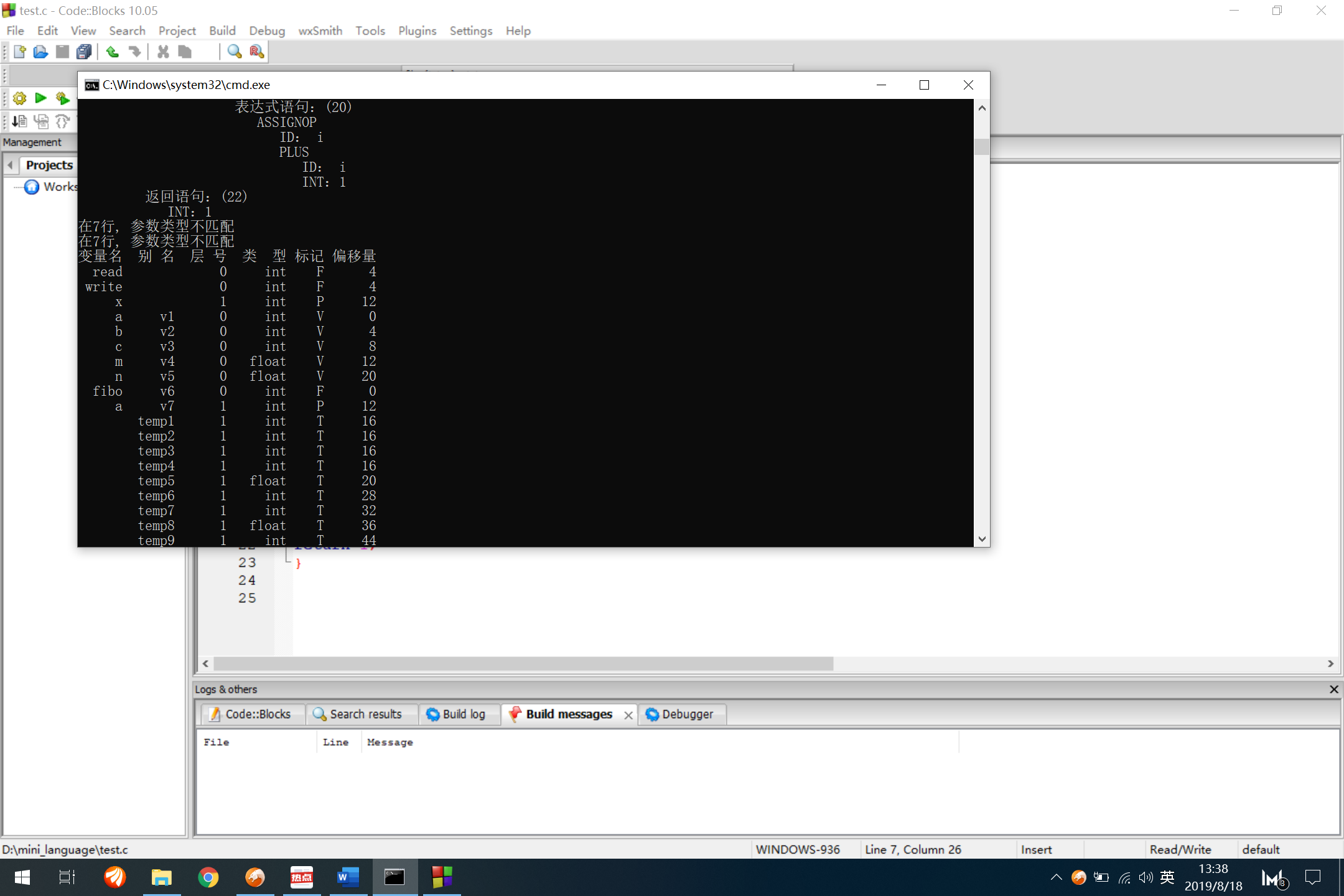
（1）每次遇到一个复合语句的结点COM\_STM，首先对LEV加1，表示准备进入一个新的作用域，为了管理这个作用域中的变量，使用栈symbol\_scope\_TX，记录该作用域变量在符号表中的起点位置，即将符号表symbolTable的栈顶位置symbolTable.index保存在栈symbol\_scope\_TX中。

（2）每次要登记一个新的符号到符号表中时，首先在symbolTable中，从栈顶向栈底方向查层号为LEV的符号，是否有和当前待登记的符号重名，是则报重复定义错误，否则使用LEV作为层号将新的符号登记到符号表中。

（3） 每次遍历完一个复合语句结点COM\_STM的所有子树，准备回到其父结点时，这时该复合语句语义分析完成，需要从符号表中删除该复合语句的变量，方法是首先symbol\_scope\_TX退栈，取出该复合语句作用域的起点，再根据这个值修改symbolTable.index，同时LEV减一，很简单地完成了符号表的符号删除操作。

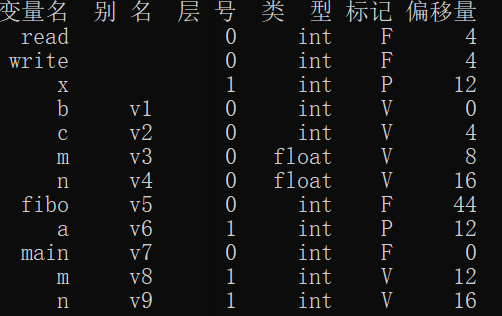
（4）符号表的查找操作，在AST的遍历过程中，分析各种表达式，遇到变量的访问时，在symbolTable中，从栈顶向栈底方向查询是否有相同的符号定义，如果全部查询完后没有找到，就是该符号没有定义；如果相同符号在符号表中有多处定义，按查找的方向可知，符合就近优先的原则。如果查找到符号后，就进一步进行语义分析，如：（1）函数调用时，根据函数名在符号表找到的是一个变量，不是函数，需要报错；（2）函数调用时，根据函数名找到这个函数，需要判断参数个数、类型是否匹配；（3）根据变量名查找的是一个函数。等等，需要做出各种检查。

对图2-4中的测试程序，当在访问函数Fibo的返回语句结点时，符号表的内容如图3-2所示。read和write为预先定义好的函数，可以直接使用。



**图3-2 符号表内容一**

接着，退出fibo的函数体的复合语句时，需要删除Fibo中的局部变量，此例中，fibo没有局部变量。接着开始访问main函数的子树，当遍历完main函数的说明语句后，符号表的添加了变量m和n，符号表中的内容如图3-3所示。



**图3-3 符号表内容二**

在语义分析编码过程中，可以随时在任意点显示符号表的内容，观察是否正确地管理好了符号的作用域。

# 4中间代码的生成

通过前面对AST遍历，完成了语义分析后，如果没有语法语义错误，就可以再次对AST进行遍历，计算相关的属性值，利用符号表，生成以三地址代码TAC作为中间语言的中间语言代码序列。

这一章中，对本实验的实现做了一些限制，假设数据类型只包含整数类型，不包含如浮点数和指针等数据类型。其它数据类型的实现，比如数组、结构，可根据实验要求进行扩充。

## 4.1 中间语言的定义

采用三地址代码TAC作为中间语言，中间语言代码的定义如表4-1所示。

**表4-1 中间代码定义**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **语法** | **描述** | **Op** | **Opn1** | **Opn2** | **Result** |
| LABEL x | 定义标号x | LABEL |  |  | x |
| FUNCTION f: | 定义函数f | FUNCTION |  |  | f |
| x := y | 赋值操作 | ASSIGN | x |  | y |
| 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 |
| GOTO x | 无条件转移 | GOTO |  |  | x |
| IF x [relop] y GOTO z | 条件转移 | [relop] | x | y | x |
| RETURN x | 返回语句 | RETURN |  |  | x |
| ARG x | 传实参x | ARG |  |  | x |
| x:=CALL f | 调用函数(有返回值) | CALL | f |  | x |
| CALL f | 调用函数(无返回值) | CALL | f |  |  |
| PARAM x | 函数形参 | PARAM |  |  | x |

三地址中间代码TAC是一个4元组，逻辑上包含（op、opn1、opn2、result），其中op表示操作类型说明，opn1和opn2表示2个操作数，result表示运算结果。后续还需要根据TAC序列生成目标代码，所以设计其存储结构时，每一部分要考虑目标代码生成时所需要的信息。

（1）运算符：表示这条指令需要完成的运算，可以用枚举常量表示，如PLUS表示双目加，JLE表示小于等于，PARAM表示形参，ARG表示实参等。

（2）操作数与运算结果：这些部分包含多种类型：整常量、实常量、标识符（如变量的别名、变量在其数据区的偏移量（外部变量给出的是在静态数据区的偏移量，局部变量、临时变量给出的是在活动记录空间的偏移量）、转移语句中的标号等。类型互不相同，所以考虑使用联合。为了明确联合中的有效成员，将操作数与运算结果设计成结构类型，包含kind，联合等几个成员，kind说明联合成员属于哪种类型，是整常量、或是实常量、或是标识符表示的别名、或是标号、或是函数名等？

（3）为了配合后续的TAC代码序列的生成，将TAC代码作为数据元素，用双向循环链表（也可以用单链表）表示TAC代码序列。

## 4.2 翻译模式

参考文献[3]中p211-p217所叙述的翻译模式，需要好好理解翻译模式表示的属性计算次序。依据属性的计算次序，在遍历AST的过程中，完成中间代码的生成。具体的方法是：在这些翻译模式中，每一个文法非终结符通常会对应AST中的一个结点。例如规则A → M..... X..... N的翻译模式：

A → M...... {X的继承属性计算}X..... N {A的综合属性计算}

对应的AST形式如图4-1所示。



**图4-1 翻译模式规则部分的AST**

在非终结符X的前面有语义属性的计算，表示在遍历到结点X的父结点A，并访问完X左边的所有子树，准备访问该结点X时，可以使用结点A以及结点X之前的结点属性，进行规则中非终结符X的语义属性计算，这里体现的是非终结符X的继承属性计算；在翻译模式中规则最后所定义的语义属性计算，表示该规则左部的非终结符A对应的子树全部遍历完成后，从N回到父结点A时，需要完成的语义属性计算，这里体现的是综合属性的计算。

为了完成中间代码的生成，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）。

为了生成中间代码序列，定义了几个函数：

**newtemp** 生成一临时变量，登记到符号表中，以**temp+序号**的形式组成的符号串作为别名，符号名称用空串的形式登记到符号表中。

**newLabel** 生成一个标号，标号命名形式为**LABEL+序号**。

**genIR** 生成一条TAC的中间代码语句。一般情况下，TAC中，涉及到2个运算对象和运算结果。如果是局部变量或临时变量，表示在运行时，其对应的存储单元在活动记录中，这时需要将其偏移量（offset）这个属性和数据类型同时带上，方便最后阶段的目标代码生成。全局变量也需要带上偏移量，确定其在静态数据区的存储单元位置。

**genLabel** 生成标号语TAC的中间代码语句。

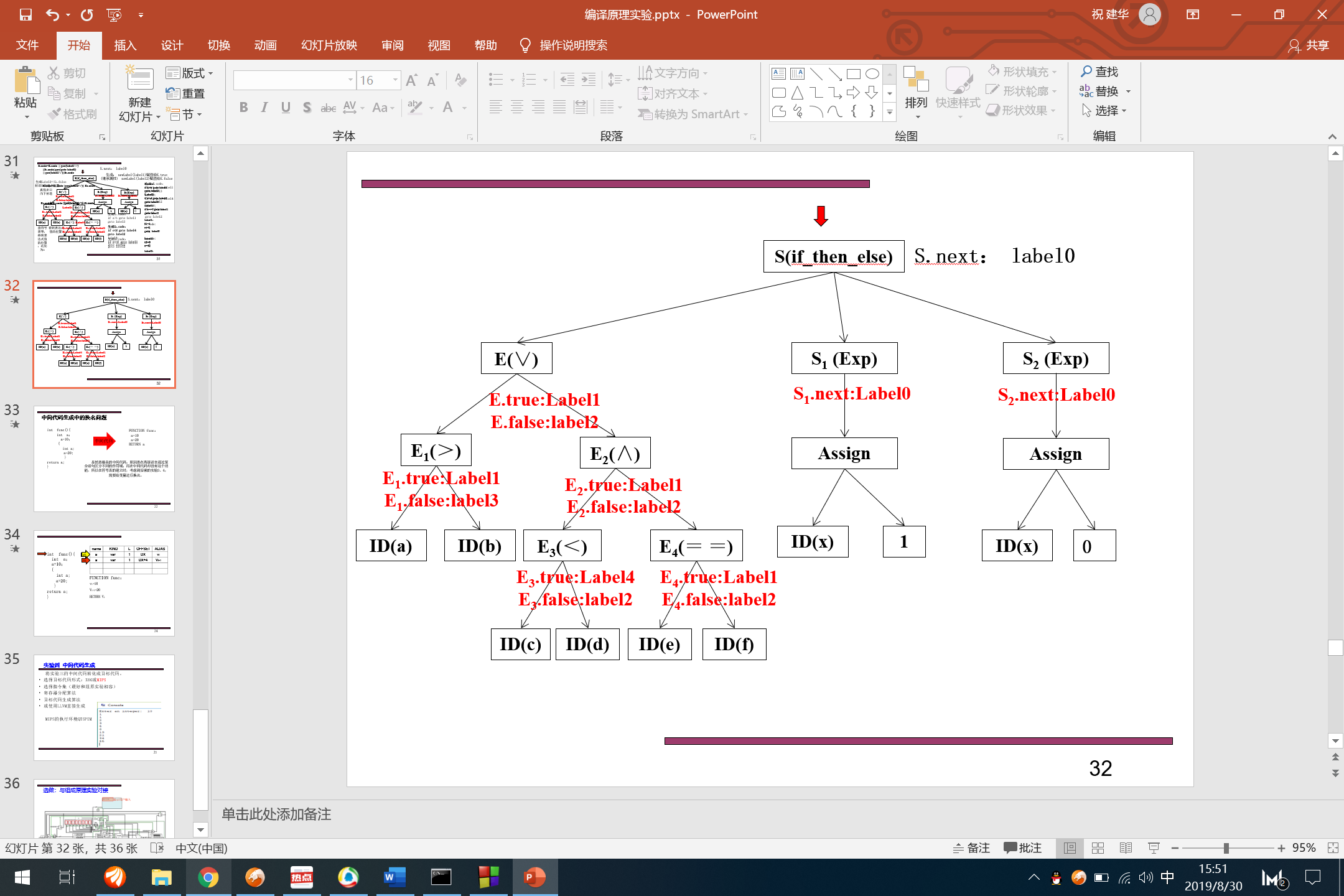
以上2个生成TAC语句的函数，在实验时，也可以合并在一起，如何处理，可自行确定。

**merge** 将多个TAC语句序列顺序连接在一起。

定义完这些属性和函数后，就需要根据翻译模式表示的计算次序，计算规则右部各个符号对应结点的代码段，再按语句的语义，将这些代码段拼接在一起，组成规则左部非终结符对应结点的代码段。

这种翻译模式表示的翻译方法，在实验时的具体体现是对抽象语法树进行遍历，在遍历过程中，完成各种属性的计算，并根据各语法成分的语义，完成中间代码的翻译。

当从函数定义结点开始，准备遍历函数体时，首先给函数体子树的根结点生成一个.Snext属性，标识函数体语句执行完成后到达的位置，接着以此为起点，遍历函数体子树，计算函数体子树中所有结点的.Snext属性。这样处理到每个语句结点时，都会有一个.Snext属性值，方便语句的翻译。如图4-2所示为一棵条件语句的抽象语法树，下面根据上述翻译模式体现出的方法，介绍在遍历过程中如何生成中间代码序列的。



**图4-2 条件语句抽象语法子树**

当对程序的抽象语法树进行遍历，到达子树根结点S的父结点时，已经由S的父节点计算出了S的继承属性.Snext，假定为label0，表示S表示的条件语句执行完后，出口是标号label0处。条件语句有3个部分：条件表达式E、if子句S1和else子句S2，对应3棵子树，需要按前根遍历的规则，依次进行遍历。

1. **条件语句的遍历：**根据语义，条件表达式E的值为真时，执行S1，否则执行S2。按遍历规则，先遍历E，再遍历S1，最后遍历S2，但在遍历E时，S1和S2的代码还未生成，无法表示E值确定后转移的目标位置。为解决这个问题，在开始访问S时，生成E的2个继承属性：E.Etrue为自动生成的标号label1、E.Efalse为标号label2，在后续连接生成S的代码时，将这2个标号分别放在S1和S2的代码前面，这样就能在E值的计算过程中，一旦确定E的值，就能方便地转移到相应用标号表示的目标位置。

当S1或S2的代码执行结束后，整个条件语句S就结束了，需要转到条件语句的出口label0处。为此在访问S结点时，还需要计算S1和S2的继承属性.Snext，都赋值为label0。

最后当三棵子树都遍历完成后，处理S的综合属性，得到S的中间代码序列：S.code=E.code || E.Etrue || S1.code || goto label0 || E.Efalse || S2.code。

1. **条件表达式E的遍历：**表达式作为控制语句中条件时，可以采用短路计算的处理方式，即在按计算次序逐步进行计算条件表达式值的过程中，一旦在某计算步骤能够确定表达式的值，就不需要完成后续的计算，直接转移到目标位置，不一定完成所有计算步骤。在图4-2中，结点E维护着2个属性：.Etrue和.Efalse，用标号的形式分别表示布尔表达式值为真或为假时转移的目标位置。

E是一个表示“逻辑或”运算的结点，按运算性质，当计算得到E1的值为真时，不需要计算E2就能确定 E的值为真，可跳过后续计算步骤，转移到E为真时的目标位置；只有当E1值为假时，才需要计算E2的值来确定E的值，这时需要一条转移语句，转移到E2的中间代码前面，开始E2的计算，为此需要自动生成一个标号label3放在E2的代码前面。于是在访问结点E时，计算E1的继承属性：E1.Etrue=E.Etrue和E1.Efalse=label3。至于E2，显然当计算得到E2的值为真时，E的值就为真，否则E的值为假，即此时，可直接由E2的值得到E的值。所以访问E时，可计算E2的继承属性：E2.Etrue=E.Etrue和E2.Efalse=E.Efalse。

采用类似方法分析E2这个“逻辑与”结点，在访问E2时，可计算E3和E4的继承属性：E3.Etrue=label4、E3.Efalse=E2.Efalse、E4.Etrue=E2.Etrue和E4.false=E2.Efalse。

处理条件运算结点时，如E1，根据E1的.Etrue和.Efalse这两个属性，生成中间代码：E1.code= if a>b goto E1.Etrue || goto E1.Efalse。类似得到E3和E4的代码。

当E2的子树遍历结束后，处理E2的综合属性，连接得到E2的中间代码序列：E2.code=E3.code || lable4 || E4.code

最后当E的所有子树遍历结束后，处理E的综合属性，得到E的中间代码序列：E.code=E1.code || lable3 || E2.code。

一般情况，对抽象语法树结点的处理，可以将结点分为若干类：执行语句、基本表达式、布尔表达式和其它结点。依据文献[3]的翻译模式，结合对各种语句、运算符的语义理解，在表4-2至表4-5给出了各类结点的中间代码翻译方法。如表4-2所示，给出了部分执行语句类结点的中间代码翻译方法。

**表4-2 语句类结点的中间代码生成**

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| COMP\_STM  T1 说明部分子树  T2 语句部分子树 | 访问到T：T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T2.code |
| IF\_THEN  T1 条件子树  T2 if子句子树 | 访问到T：T1.Etrue=newLabel, T1.Efalse= T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T1.Etrue ||T2.code |
| IF\_THEN\_ELSE  T1 条件子树  T2 if子句子树  T3 else子句子树 | 访问到T： T1.Etrue=newLabel,T1.Efalse=T.Snext  T1.Snext= T2.Snext=T.Snext  访问T的所有子树后：  T.code=T1.code || T1.Etrue || T2.code || goto T.Enext  || T1.Efalse || T3.code |
| WHILE  T1 条件子树  T2 if子句子树 | 访问到T：T1.Etrue=newLabel, T1.Efalse=T.Snext,  T2.Snext= newLabel;  访问T的所有子树后：  T.code=T2.Snext || T1.code || T1.Etrue || T2.code || goto T2.Snext |
| STM\_LIST  T1 语句1子树  T2 语句2子树（可空） | 访问到T：if(T2非空) {T1.Snext=newLabel , T2.Snext=T.Snext;}  else T1.Snext=S.next;  访问T的所有子树后：  if (T2为空) T.code=T1.code  else T.code=T1.Snext || T1.Snext || T2.code |
| EXP\_STM  T1 表达式子树 | 访问到T：T1.Snext=T.Snext;  访问T的所有子树后：T.code=T1.code |
| RETURN  T1 表达式子树（可空） | 访问到T：T1.Snext=T.Snext;  访问T的所有子树后：  if (T1非空) T.code=T1.code|| return T1.alias  else T.code=T1.code|| return |

**注：**① T表示当前结点，T1、T2……表示T的第1个孩子、第2个孩子、……；

② || 表示连接操作，实现将中间代码序列的连接；

③ Ti.alias表示根据结点Ti的place属性得到符号表中对应的变量别名或临时变量名。

表达式的计算，分为两种类型，第一种是基本表达式的计算，第二种是布尔表达式的计算。通常第一种可以构造表达式语句，第二种是在控制语句（条件语句和循环语句）中作为控制条件，两种含义表达式的中间代码翻译方法，既可以分开进行处理，也可以合并在一起。基本表达式的翻译，需要按计算次序，完成表达式中的所有计算操作步骤，最后得到表达式的值。如表4-3所示，给出了基本表达式类结点的中间代码翻译方法。在翻译中，注意每个结点对应一个值，需要一个变量或临时变量保存该值，这样对于常量结点，可生成一个临时变量保存

**表4-3 基本表达式类结点的中间代码生成**

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| INT  其它如FLOAT类的结点按类似方法处理 | ti=newtemp, ti在符号表的入口赋值给T.place。后续可通过T.alias读取该值。  T.code为： ti = INT的值 |
| ID | ID在符号表中的入口赋值给T.place，代码为空 |
| ASSIGNOP  T1 左值表达式子树  T2 左值表达式子树 | 访问到T： T. palce=T1.place  访问T的所有子树后：T.code=T1.code || T2.code || T1.alias= T2.alias |
| OP 算术运算符。  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place  访问T的所有子树后：  T.code=T1.code || T2.code || ti=T1.alias OP T2.alias |
| UMINUS  T1 操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place  访问T的所有子树后：T.code=T1.code || ti=- T1.alias |
| RELOP 关系运算符  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code  || if T1.alias RELOP T2.alias goto label1  || ti=0 || goto label2 || label1： || ti=1 || label2： |
| AND  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code || ti=T1.alias \* T2.alias  || if ti==0 goto label1 || ti=1 || label1： |
| OR  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code=T1.code || T2.code || ti=0  || if T1.alias ==0 goto label1 || ti=1 || goto label2 || label1：  || if T2.alias ==0 goto label2 || ti=1 || label2： |
| NOT  T1 操作数子树 | ti=newtemp, ti在符号表的入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的子树后：  T.code= if T1.alias ==0 goto label1  || ti=0 || goto label2 || label1：|| ti=1 || label2： |
| FUNC\_CALL  T1 实参列表子树 | ti=newtemp, ti在符号表的入口赋值给T.place  T.code=T1.code;  访问T的子树，从上至下依次对每个ARGS实参结点T0，完成实参处理。  T.code= T. code || ARG T01.alias  这里T01表示T0的第一个孩子，访问T的子树后：  T.code= T.code || ti=CALL 函数名 |

常量值，对于运算符结点（除赋值运算外），也需要一个临时变量保存运算结果。通过属性.palce保存变量或临时变量在符号表中的位置。

布尔表达式用于控制语句中，不同于基本表达式，布尔表达式值的计算，采用短路语句处理方式。这时面临的一个问题，就是同样形式的表达式，如何确定是按基本表达式进行处理，还是按布尔表达式进行处理？在结点中维护2个属性：.Etrue和.Efalse，分别确定布尔表达式值为真或为假时转移的目标位置，初始值都是空串。如表4-1所示，当表达式作为控制语句的条件时，都会在其控制语句结点处给条件表达式子树的根结点属性.Etrue和.Efalse赋值，所以在处理表达式结点时，如果属性.Etrue和.Efalse的值为空就按基本表达式处理，否则按布尔表达式处理。

布尔表达式语句类结点的中间代码的翻译方法,除逻辑与、或、非和关系运算外，其它的可以由基本表达式的翻译进行一下扩展：即完成了基本表达式的计算后，如果结点的.Etrue 和.Efalse属性值不为空，就增加语句，当结点值不等于0时，转移到.Etrue位置，否则转移到.Efalse。如表4-4所示为控制语句中布尔表达式的中间代码翻译方法。

**表4-4 控制语句布尔表达式语句结点的中间代码生成**

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| INT  其它如FLOAT类的结点按类似方法处理 | if (T.Etrue==””) 按基本表达式处理  else if (INT的值) T.code= goto T.Etrue  else T.code= goto T.Efalse |
| ID | ID在符号表中的入口赋值给T.place  if (T.Etrue==””) 按基本表达式处理  else T.code= if T.alias!=0 goto T.Etrue || goto T.Efalse |
| ASSIGNOP  T1 左值表达式子树  T2 左值表达式子树 | T.palce=T1.place  访问T的所有子树后：  T.code=T1.code || T2.code || T1.alias= T2.alias  if (T.true!=””)  T.code=T.code || if T1.alias!=0 goto T.Etrue || goto T.Efalse |
| OP 算术运算符。  T1 第一操作数子树  T2 第二操作数子树 | ti=newtemp, ti入口赋值给当前结点T.place  访问T的所有子树后：  T.code =T1.code || T2.code || ti=T1.alias OP T2.alias  if (T.true!=””)  T.code = T.code || if ti!=0 goto T.Etrue || goto T.Efalse |
| UMINUS  T1 操作数子树 | ti=newtemp, ti入口赋值给当前结点place  访问T的所有子树后：  T.code = T1.code || ti=- T1.alias  if (T.true!=””)  T.code=T.code || if ti!=0 goto T.Etrue || goto T.Efalse |
| RELOP 关系运算符  T1 第一操作数子树  T2 第二操作数子树 | if (T.true==””) ti=newtemp, ti入口赋值给T.place，  Label1=newLabel，Label2=newLabel。  访问T的所有子树后：  T.code = T1.code || T2.code  if (T.true!=””)  T.code=T.code || if T1.alias RELOP T2.alias goto T.Etrue  || goto T.Efalse  else T.code=T.code || if T1.alias RELOP T2.alias goto label1  || ti=0 || goto label2 || label1：|| ti=1 || label2： |
| AND  T1 第一操作数子树  T2 第二操作数子树 | if (T.Etrue==””) 按基本表达式处理  else T1.Etrue= newLabel, T2.Etrue= T.Etrue,  T1.Efalse= T2.Efalse =T.Efalse;  T.code=T1.code || T1.Etrue || T2.code |
| OR  T1 第一操作数子树  T2 第二操作数子树 | if (T.Etrue==””) 按基本表达式处理  else T1.Etrue=T2.Etrue= T.Etrue,T1.Efalse= newLabel,  T2..Efalse =T.Efalse;  T.code=T1.code || T1.Efalse || T2.code |
| NOT  T1 操作数子树 | f (T.Etrue==””) 按基本表达式处理  else T1.Etrue=T.Efalse, T1.Efalse =T.Etrue;  T.code=T1.code： |
| FUNC\_CALL  T1 实参列表子树 | ti=newtemp, ti入口赋值给T.place  T.code=T1.code;  访问T的子树，从上至下依次对每个ARGS结点T0，完成实参处理。  T.code= T. code || ARG T01.alias  这里T01表示T0的第一个孩子，访问T的子树后：  T.code= T.code || ti=CALL 函数名  f (T.Etrue!=””) T.code=T.code || if ti!=0 goto T.Etrue || goto T.Efalse |

其它类结点的翻译如表4-5所示。

**表4-5 其它类结点的中间代码生成**

|  |  |
| --- | --- |
| **当 前 结 点 类 型** | **翻 译 动 作** |
| FUNC\_DEF  T1 返回值类型  T2 函数名与参数  T3 函数体 | 访问T时：T3.Snext=newLabel  访问T的所有子树后:  T.code=T2.code || T3.code || T3.Snext |
| FUNC\_DEC  T1 参数列表（可空） | 访问T的所有子树后:  T.code=FUNCTION 函数名  if (T1非空) T.code=T.code || T1.code |
| PARAM\_LIST  T1 形参说明子树  T2形参列表子树（可空） | 访问T的所有子树后:  T.code=T1.code  if (T2非空) T.code=T.code || T2.code |
| PARAM\_DEC  T1 形参类型  T2形参名 | 访问T的所有子树后:  T.code=PARAM T2.alias |
| ARGS  T1 实参子树  T2 实参列表子树（可空） | 访问T的所有子树后：  if（T2==NULL）T.code= T1.code  else T.code= T1.code || T2.code |

根据上述介绍的中间代码翻译模式，对图2-4中的测试程序进行处理，生成的中间代码序列如下。

**FUNCTION fibo :**

**PARAM v7**

**temp1 := #1**

**IF v7 == temp1 GOTO label3**

**GOTO label4**

**LABEL label4 :**

**temp2 := #2**

**IF v7 == temp2 GOTO label3**

**GOTO label2**

**LABEL label3 :**

**temp3 := #1**

**RETURN temp3**

**LABEL label2 :**

**temp4 := #1**

**temp5 := v7 - temp4**

**ARG temp5**

**temp6 := CALL fibo**

**temp7 := #2**

**temp8 := v7 - temp7**

**ARG temp8**

**temp9 := CALL fibo**

**temp10 := temp6 + temp9**

**RETURN temp10**

**LABEL label1 :**

**FUNCTION main :**

**temp11 := CALL read**

**v9 := temp11**

**temp12 := #1**

**v11 := temp12**

**LABEL label10 :**

**IF v11 <= v9 GOTO label9**

**GOTO label8**

**LABEL label9 :**

**ARG v11**

**temp13 := CALL fibo**

**v10 := temp13**

**ARG v10**

**CALL write**

**temp14 := #1**

**temp15 := v11 + temp14**

**v11 := temp15**

**GOTO label10**

**LABEL label8 :**

**temp16:= #1**

**RETURN temp16**

**LABEL label5 :**

# 5目标代码的生成

这部分的实验要完成将TAC的指令序列转换成目标代码。作为实验指导的目的，本文中对实现做了一些限制，假设数据类型只包含整数类型，不包含如浮点数、数组、结构和指针等其它数据类型，目标语言为汇编语言。这样做的目的就是尽可能简单地实现目标代码的生成并能运行程序观察运行结果，完成了一个可运行的编译程序。实验时，需要根据自己定义的语言，预期目标进行实现完成，比如数组、结构等。

## 5.1 目标语言的指令定义

目标语言可选定MIPS32指令序列， TAC指令和MIPS32指令的对应关系如表5-1所示。其中reg(x)表示变量x所分配的寄存器。

**表5-1 中间代码与MIPS32指令对应关系**

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| LABEL x | x： |
| x :=#k | li reg(x),k |
| x := y | move reg(x), reg(y) |
| x := y + z | add reg(x), reg(y) , reg(z) |
| x := y - z | sub reg(x), reg(y) , reg(z) |
| x := y \* z | mul reg(x), reg(y) , reg(z) |
| x := y / z | div reg(y) , reg(z)  mflo reg(x) |
| GOTO x | j x |
| RETURN x | move $v0, reg(x)  jr $ra |
| IF x==y GOTO z | beq reg(x),reg(y),z |
| IF x!=y GOTO z | bne reg(x),reg(y),z |
| IF x>y GOTO z | bgt reg(x),reg(y),z |
| IF x>=y GOTO z | bge reg(x),reg(y),z |
| IF x<y GOTO z | blt reg(x),reg(y),z |
| IF x<=y GOTO z | ble reg(x),reg(y),z |
| X:=CALL f | jal f  move reg(x),$v0 |

当把TAC翻译成MIPS的汇编程序后，可以选择在下列虚拟机上运行：

1. SPIM Simulator，SPIM Simulator有两个版本：

命令行版本，在linux环境下使用命令sudo apt-get install spim进行安装；

GUI版本（QTSPIM），该版本可在Windows, Mac OS X,或Linux环境下运行，在<http://pages.cs.wisc.edu/~larus/spim.html>下载。

安装及使用参见文献[2]。

1. MARS（MIPS Assembler and Runtime Simulator），MARS是密苏里州立大学开发的一个轻量级的交互式开发环境（IDE），用于使用MIPS汇编语言进行编程。可在[http://courses.missouristate.edu/kenvollmar/mars](https://links.jianshu.com/go?to=http%3A%2F%2Fcourses.missouristate.edu%2Fkenvollmar%2Fmars%2F)下载。MARS是一个免安装的环境，下载的是一个可执行文件，直接运行就进入了环境界面。
2. 使用组成原理课程上实现CPU的指令集，最终编译成机器码在自己的CPU上运行。

注意有些环境，如MARS，要求MIPS汇编程序的入口函数要放在最前面，可以考虑目标代码生成时，把main函数调整到最前面，或增加一个放在最前面的入口函数（假定为main0），在入口函数中调用main函数。对应代码段为：

.globl main0

.text

main0:

addi $sp, $sp, -main函数的活动记录大小

jal main

li $v0,10 #使用系统调用，终止程序

syscall

## 5.2 寄存器的分配

有关寄存器等的详细描述参见文献[2]。在目标生成阶段，一个很重要的工作就是寄存器的分配，在文献[2][3]中给出了不少算法可供参考，其中最为简单的就是朴素的寄存器分配算法，效率最低，也最容易实现；对于一个基本块采用的局部寄存器分配算法，实现起来相对不是太难，且效率上有一定提升；其它的算法基本上都要涉及到数据流的分析，效率会提升很多，但在实验中，由于学时的原因，对其相关的理论部分，数据流的分析介绍较少，这样实现起来会有较大难度。

这样在实验时，不对寄存器的分配算法做任何具体要求。但在时间许可的前提下，鼓励通过自学完成寄存器的分配算法。

## 5.3 目标代码的生成

当选择朴素的寄存器分配方案后，目标代码生成时，每当运算操作时，都需要将操作数读入到寄存器中，运算结束后将结果写到对应的单元。由于选择朴素的寄存器分配，只会用到几个寄存器，这里约定操作数使用$t1和$t2，运算结果使用$t3，翻译的方法如表5-2所示。

**表5-2 朴素寄存器分配的翻译**

|  |  |
| --- | --- |
| **中间代码** | **MIPS32指令** |
| x :=#k | li $t3,k  sw $t3, x的偏移量($sp) |
| x := y | lw $t1, y的偏移量($sp)  move $t3,$t1  sw $t3, x的偏移量($sp) |
| x := y + z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  add $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y - z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  sub $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y \* z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  sw $t3, x的偏移量($sp) |
| x := y / z | lw $t1, y的偏移量($sp)  lw $t2, z的偏移量($sp)  mul $t3,$t1,$t2  div $t1,$t2  mflo $t3  sw $t3, x的偏移量($sp) |
| RETURN x | move $v0, x的偏移量($sp)  jr $ra |
| IF x==y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  beq $t1,$t2,z |
| IF x!=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bne $t1,$t2,z |
| IF x>y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bgt $t1,$t2,z |
| IF x>=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  bge $t1,$t2,z |
| IF x<y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  blt $t1,$t2,z |
| IF x<=y GOTO z | lw $t1, x的偏移量($sp)  lw $t2, y的偏移量($sp)  blt $t1,$t2,z |
| X:=CALL f |  |

对于函数调用X:=CALL f，需要完成开辟活动记录的空间、参数的传递和保存返回地址等，函数调用返回后，需要恢复返回地址，读取函数返回值以及释放活动记录空间。活动记录的空间布局没有一个统一的标准，可根据自己的理解保存好数据，并能正确使用即可。

通常，使用4个寄存器完成参数的传递，多余4个的参数使用活动记录空间，这里做了简单处理，所有参数都使用活动记录空间。具体步骤：

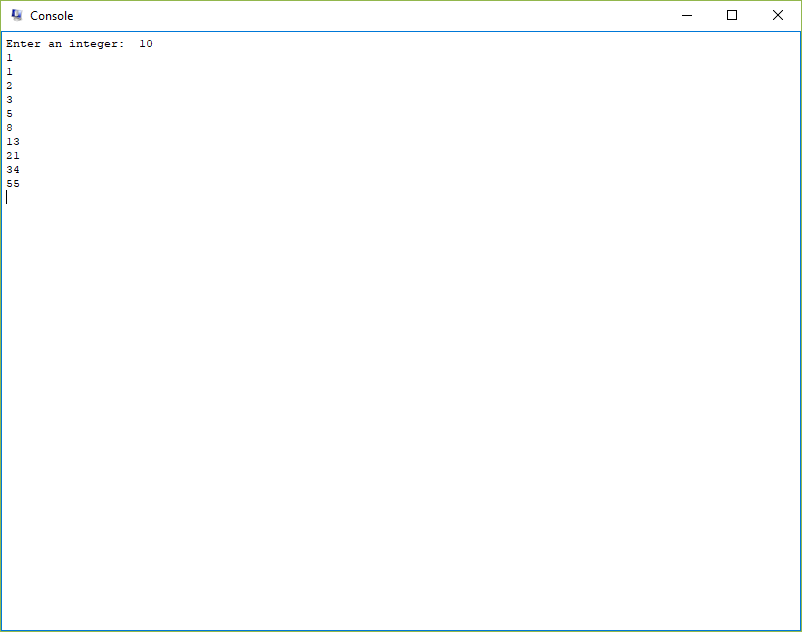
1. 首先根据符号表中函数定义项得到该函数活动记录的大小，开辟活动记录空间和保存返回地址。思考一下main函数的活动记录如何处理？
2. 根据函数定义中的参数个数paramnum，即在X:=CALL f之前有paramnum个ARG形式的中间代码，可获得各个实参值所存放的单元，取出后送到形式参数的单元中。

（3） 使用jal f 转到函数f处

（4） 释放活动记录空间和恢复返回地址。

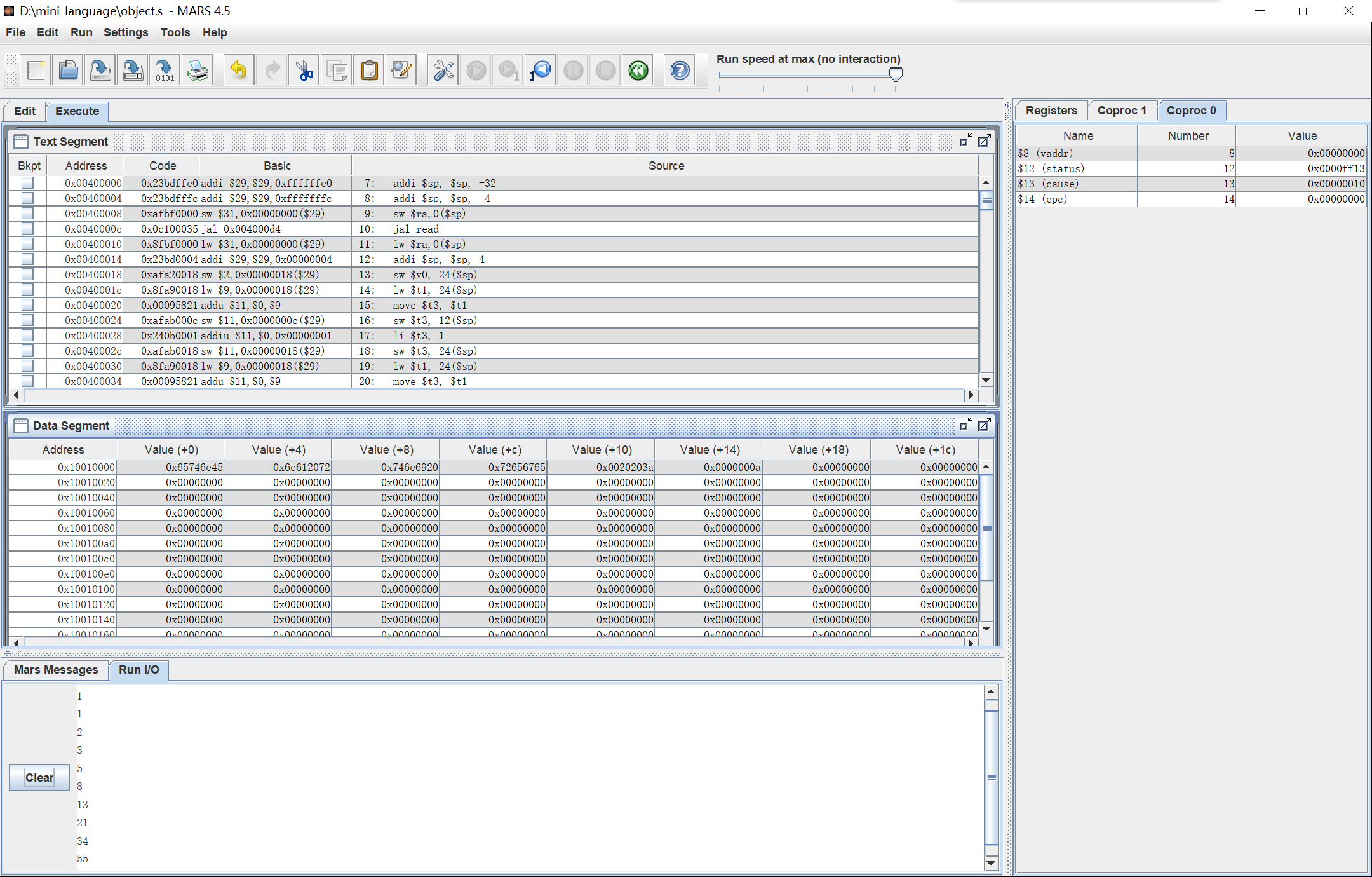
（5） 使用sw $v0, X的偏移量($sp) 获取返回值送到X的存储单元中。

按上述方式完成测试程序的目标代码的生成，在QTSPIM中运行的运行结果如图5-1所示。



**图5-1 测试程序运行结果（QTSPIM）**

在MARS中运行的运行结果如图5-2所示。



**图5-2 测试程序运行结果（MARS）**

# 附录1： 词法分析的程序文件lex.l

%{

#include "parser.tab.h"

#include "string.h"

#include "def.h"

int yycolumn=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;

float type\_float;

char type\_id[32];

struct node \*ptr;

} YYLVAL;

#define YYSTYPE YYLVAL

%}

%option yylineno

id [A-Za-z][A-Za-z0-9]\*

int [0-9]+

float ([0-9]\*\.[0-9]+)|([0-9]+\.)

%%

{int} {yylval.type\_int=atoi(yytext); return INT;}

{float} {yylval.type\_float=atof(yytext); return FLOAT;}

"int" {strcpy(yylval.type\_id, yytext);return TYPE;}

"float" {strcpy(yylval.type\_id, yytext);return TYPE;}

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

{id} {strcpy(yylval.type\_id, yytext); return ID;/\*由于关键字的形式也符合标识符的规则，所以把关键字的处理全部放在标识符的前面，优先识别\*/}

";" {return SEMI;}

"," {return COMMA;}

">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type\_id, yytext);;return RELOP;}

"=" {return ASSIGNOP;}

"+" {return PLUS;}

"-" {return MINUS;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"{" {return LC;}

"}" {return RC;}

[\n] {yycolumn=1;}

[ \r\t] {}

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

/\*作为实验内容，还需要考虑识别出2种形式的注释注释部分时，直接舍弃 \*/

%%

/\* 和bison联用时，不需要这部分

void main()

{

yylex();

return 0;

}

\*/

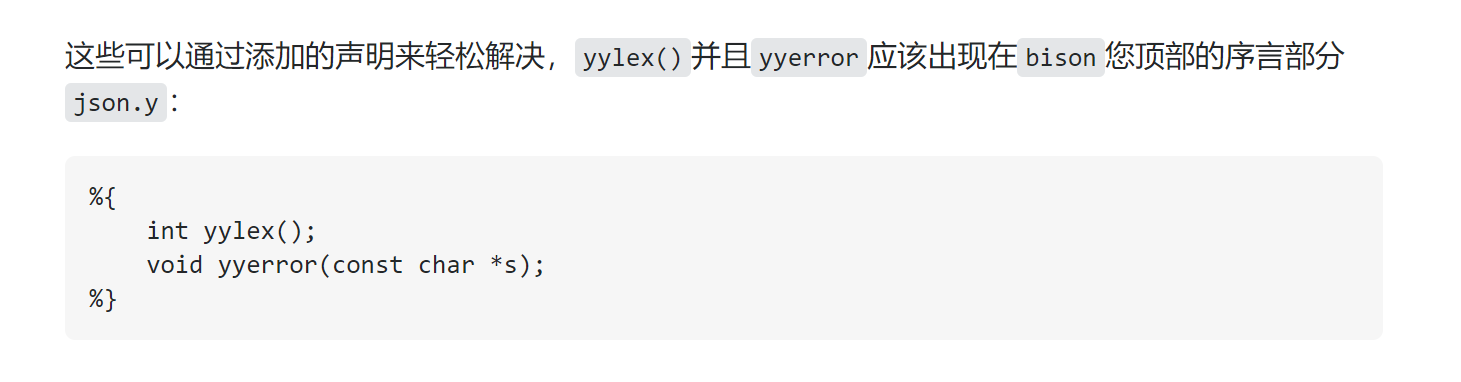
int yywrap()

{

return 1;

}

# 附录2： 语法分析的程序文件parser.y



%error-verbose

%locations

%{

#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);

%}

%union {

int type\_int;

float type\_float;

char type\_id[32];

struct ASTNode \*ptr;

};

// %type 定义非终结符的语义值类型

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args CaseStmtList0 CaseStmtList

//% token 定义终结符的语义值类型

%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 RELOP TYPE /\*指定ID,RELOP, TYPE 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 DPLUS LP RP LC RC SEMI COMMA /\*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 ASSIGNOP AND OR NOT IF ELSE WHILE RETURN FOR SWITCH CASE COLON DEFAULT

/\*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

%left ASSIGNOP

%left OR

%left AND

%left RELOP

%left PLUS MINUS

%left STAR DIV

%right UMINUS NOT DPLUS

%nonassoc LOWER\_THEN\_ELSE

%nonassoc ELSE

%%

program: ExtDefList { 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

| error SEMI {$$=NULL;}

;

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

;

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

;

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);}

;

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);}

;

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: 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 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");}

| 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);}

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

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

;

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

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

;

%%

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

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

if (!yyin) return 0;

yylineno=1;

yyparse();

return 0;

}

#include<stdarg.h>

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");

}

# 附录3： 有关定义文件def.h

#include "stdio.h"

#include "stdlib.h"

#include "string.h"

#include "stdarg.h"

#include "parser.tab.h"

#define MAXLENGTH 200

#define DX 3\*sizeof(int) /\*活动记录控制信息需要的单元数，这个根据实际系统调整\*/

//以下语法树结点类型、三地址结点类型等定义仅供参考，实验时一定要根据自己的理解来定义

int LEV; //层号

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; //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

};

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;

};

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

};

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

};

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;

struct ASTNode \* mknode(int num,int kind,int pos,...);

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

void semantic\_Analysis(struct ASTNode \*T);

void boolExp(struct ASTNode \*T);

void Exp(struct ASTNode \*T);

void objectCode(struct codenode \*head);

# 附录4： 抽象语法树的建立与显示ast.c

#include "def.h"

#include "parser.tab.h"

struct ASTNode \* mknode(int num,int kind,int pos,...){

struct ASTNode \*T=(struct ASTNode \*)calloc(sizeof(struct ASTNode),1);

int i=0;

T->kind=kind;

T->pos=pos;

va\_list pArgs = NULL;

va\_start(pArgs, pos);

for(i=0;i<num;i++)

T->ptr[i]= va\_arg(pArgs, struct ASTNode \*);

while (i<4) T->ptr[i++]=NULL;

va\_end(pArgs);

return T;

}

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+3); //显示外部变量类型

printf("%\*c变量名：\n",indent+3,' ');

display(T->ptr[1],indent+6); //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+3); //显示函数返回类型

display(T->ptr[1],indent+3); //显示函数名和参数

display(T->ptr[2],indent+3); //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+3); //显示函数参数列表

}

else printf("%\*c无参函数\n",indent+3,' ');

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+3);

break;

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

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

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+3,' ');

display(T->ptr[0],indent+6); //显示定义部分

printf("%\*c复合语句的语句部分：\n",indent+3,' ');

display(T->ptr[1],indent+6); //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("%\*c循环语句：(%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 IF\_THEN: printf("%\*c条件语句(IF\_THEN)：(%d)\n",indent,' ',T->pos);

printf("%\*c条件：\n",indent+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //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+3,' ');

display(T->ptr[0],indent+6); //显示条件

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

display(T->ptr[1],indent+6); //显示if子句

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

display(T->ptr[2],indent+6); //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+3); //显示变量类型

display(T->ptr[1],indent+3); //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+6,' ',T0->ptr[0]->type\_id);

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

{

printf("%\*c %s ASSIGNOP\n ",indent+6,' ',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

}

T0=T0->ptr[1];

}

break;

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

break;

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

break;

case FLOAT: printf("%\*cFLOAT：%f\n",indent,' ',T->type\_float);

break;

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

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

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

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

break;

case NOT:

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

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

break;

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

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

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

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个实际参数表达式：\n",indent,' ',i++);

display(T0,indent+3);

T=T->ptr[1];

}

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

// display(T,indent+3);

printf("\n");

break;

}

}

}

# 附录5： 语义分析与中间代码生成

/\* 在遍历语法树中，完成了符号表的创建、删除等操作，符号表使用的是一个顺序表，实际操作中还可以用hash表等不同形式。注意不同的作用域符号表的开始和结束删除的时机。也可以在静态语义分析过程中，为每个作用域建立一张符号表。等后续中间代码生成阶段，用一个栈来管理，每当进入一个作用域，将改作用域的符号表（指针）入栈，退出作用域是，退栈。

这段程序是一个不完整的代码，也未经过严格调试，只具有下列功能的一部分。

1. 检查变量重复定义；

2. 未定义的变量使用；

3. 变量类型的匹配；

4. 计算外部变量在数据区的偏移量；局部变量在活动记录中的偏移量；

5. 中间代码的生成。

此段程序仅作参考，并且把语义分析与中间代码混合在了一起，实验中需要按功能区分开来，第一次遍历只完成语义分析，第二次完成中间代码生成。希望阅读后，能体会在遍历AST的过程中如何完成属性计算，知道从什么地方下手，并按自己的设计思路设置必要的属性，进行语义分析、中间代码生成。

有关属性的计算，中间代码的生成，可以用播放的方式参考PPT:"语法树的遍历（符号表与中间代码）"以及教材第8章课件 \*/

#include "def.h"

#define DEBUG 1

char \*strcat0(char \*s1,char \*s2){

static char result[10];

strcpy(result,s1);

strcat(result,s2);

return result;

}

char \*newAlias() {

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("v",s);

}

char \*newLabel() {

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("label",s);

}

char \*newTemp(){

static int no=1;

char s[10];

itoa(no++,s,10);

return strcat0("temp",s);

}

//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;

}

//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;

}

//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;

}

//合并多个中间代码的双向循环链表，首尾相连

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;

}

//输出中间代码

void prnIR(struct codenode \*head){

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

struct codenode \*h=head;

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->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);

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

switch (h->op) {

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

break;

case PLUS:

case MINUS:

case STAR:

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

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

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

break;

case CALL: if (!strcmp(opnstr1,"write"))

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

else

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

break;

case RETURN: if (h->result.kind)

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

else

printf(" RETURN\n");

break;

}

h=h->next;

} while (h!=head);

}

void semantic\_error(int line,char \*msg1,char \*msg2){

//这里可以只收集错误信息，最后一次显示

printf("在%d行,%s %s\n",line,msg1,msg2);

}

void prn\_symbol(){ //显示符号表

int i=0;

printf("%6s %6s %6s %6s %4s %6s\n","变量名","别 名","层 号","类 型","标记","偏移量");

for(i=0;i<symbolTable.index;i++)

printf("%6s %6s %6d %6s %4c %6d\n",symbolTable.symbols[i].name,\

symbolTable.symbols[i].alias,symbolTable.symbols[i].level,\

symbolTable.symbols[i].type==INT?"int":"float",\

symbolTable.symbols[i].flag,symbolTable.symbols[i].offset);

}

int searchSymbolTable(char \*name) {

int i,flag=0;

for(i=symbolTable.index-1;i>=0;i--){

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

flag=1;

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

continue; //跳过前面函数的形式参数表项

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

}

return -1;

}

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

}

//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

}

void ext\_var\_list(struct ASTNode \*T){ //处理变量列表

int rtn,num=1;

switch (T->kind){

case EXT\_DEC\_LIST:

T->ptr[0]->type=T->type; //将类型属性向下传递变量结点

T->ptr[0]->offset=T->offset; //外部变量的偏移量向下传递

T->ptr[1]->type=T->type; //将类型属性向下传递变量结点

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

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

ext\_var\_list(T->ptr[0]);

ext\_var\_list(T->ptr[1]);

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

break;

case ID:

rtn=fillSymbolTable(T->type\_id,newAlias(),LEV,T->type,'V',T->offset); //最后一个变量名

if (rtn==-1)

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

else T->place=rtn;

T->num=1;

break;

}

}

int match\_param(int i,struct ASTNode \*T){

int j,num=symbolTable.symbols[i].paramnum;

int type1,type2,pos=T->pos;

T=T->ptr[0];

if (num==0 && T==NULL) return 1;

for (j=1;j<=num;j++) {

if (!T){

semantic\_error(pos,"", "函数调用参数太少!");

return 0;

}

type1=symbolTable.symbols[i+j].type; //形参类型

type2=T->ptr[0]->type;

if (type1!=type2){

semantic\_error(pos,"", "参数类型不匹配");

return 0;

}

T=T->ptr[1];

}

if (T){ //num个参数已经匹配完，还有实参表达式

semantic\_error(pos,"", "函数调用参数太多!");

return 0;

}

return 1;

}

void boolExp(struct ASTNode \*T){ //布尔表达式，参考文献[2]p84的思想

struct opn opn1,opn2,result;

int op;

int rtn;

if (T)

{

switch (T->kind) {

case INT:

break;

case FLOAT:

break;

case ID:

break;

case RELOP: //处理关系运算表达式,2个操作数都按基本表达式处理

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

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

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

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;

result.kind=ID; strcpy(result.id,T->Etrue);

if (strcmp(T->type\_id,"<")==0)

op=JLT;

else if (strcmp(T->type\_id,"<=")==0)

op=JLE;

else if (strcmp(T->type\_id,">")==0)

op=JGT;

else if (strcmp(T->type\_id,">=")==0)

op=JGE;

else if (strcmp(T->type\_id,"==")==0)

op=EQ;

else if (strcmp(T->type\_id,"!=")==0)

op=NEQ;

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

T->code=merge(4,T->ptr[0]->code,T->ptr[1]->code,T->code,genGoto(T->Efalse));

break;

case AND:

case OR:

if (T->kind==AND) {

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

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

}

else {

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

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

}

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

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

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

boolExp(T->ptr[0]);

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

boolExp(T->ptr[1]);

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

if (T->kind==AND)

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

else

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

break;

case NOT: strcpy(T->ptr[0]->Etrue,T->Efalse);

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

boolExp(T->ptr[0]);

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

break;

}

}

}

void Exp(struct ASTNode \*T)

{//处理基本表达式，参考文献[2]p82的思想

int rtn,num,width;

struct ASTNode \*T0;

struct opn opn1,opn2,result;

if (T)

{

switch (T->kind) {

case ID: //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 {

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

}

break;

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=4;

break;

case ASSIGNOP:

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

semantic\_error(T->pos,"", "赋值语句需要左值");

}

else {

Exp(T->ptr[0]); //处理左值，例中仅为变量

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

Exp(T->ptr[1]);

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

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

T->code=merge(2,T->ptr[0]->code,T->ptr[1]->code);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[1]->place].alias);//右值一定是个变量或临时变量

opn1.offset=symbolTable.symbols[T->ptr[1]->place].offset;

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

}

break;

case AND: //按算术表达式方式计算布尔值，未写完

case OR: //按算术表达式方式计算布尔值，未写完

case RELOP: //按算术表达式方式计算布尔值，未写完

T->type=INT;

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

Exp(T->ptr[0]);

Exp(T->ptr[1]);

break;

case PLUS:

case MINUS:

case STAR:

case DIV: T->ptr[0]->offset=T->offset;

Exp(T->ptr[0]);

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

Exp(T->ptr[1]);

//判断T->ptr[0]，T->ptr[1]类型是否正确，可能根据运算符生成不同形式的代码，给T的type赋值

//下面的类型属性计算，没有考虑错误处理情况

if (T->ptr[0]->type==FLOAT || T->ptr[1]->type==FLOAT)

T->type=FLOAT,T->width=T->ptr[0]->width+T->ptr[1]->width+4;

else T->type=INT,T->width=T->ptr[0]->width+T->ptr[1]->width+2;

T->place=fill\_Temp(newTemp(),LEV,T->type,'T',T->offset+T->ptr[0]->width+T->ptr[1]->width);

opn1.kind=ID; strcpy(opn1.id,symbolTable.symbols[T->ptr[0]->place].alias);

opn1.type=T->ptr[0]->type;opn1.offset=symbolTable.symbols[T->ptr[0]->place].offset;

opn2.kind=ID; strcpy(opn2.id,symbolTable.symbols[T->ptr[1]->place].alias);

opn2.type=T->ptr[1]->type;opn2.offset=symbolTable.symbols[T->ptr[1]->place].offset;

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

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

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

T->width=T->ptr[0]->width+T->ptr[1]->width+(T->type==INT?4:8);

break;

case NOT: //未写完整

break;

case UMINUS://未写完整

break;

case FUNC\_CALL: //根据T->type\_id查出函数的定义，如果语言中增加了实验教材的read，write需要单独处理一下

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

if (rtn==-1){

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

break;

}

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

semantic\_error(T->pos,T->type\_id, "不是一个函数");

break;

}

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

width=T->type==INT?4:8; //存放函数返回值的单数字节数

if (T->ptr[0]){

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

Exp(T->ptr[0]); //处理所有实参表达式求值，及类型

T->width=T->ptr[0]->width+width; //累加上计算实参使用临时变量的单元数

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

}

else {T->width=width; T->code=NULL;}

match\_param(rtn,T); //处理所有参数的匹配

//处理参数列表的中间代码

T0=T->ptr[0];

while (T0) {

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

result.offset=symbolTable.symbols[T0->ptr[0]->place].offset;

T->code=merge(2,T->code,genIR(ARG,opn1,opn2,result));

T0=T0->ptr[1];

}

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

opn1.kind=ID; strcpy(opn1.id,T->type\_id); //保存函数名

opn1.offset=rtn; //这里offset用以保存函数定义入口,在目标代码生成时，能获取相应信息

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

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

T->code=merge(2,T->code,genIR(CALL,opn1,opn2,result)); //生成函数调用中间代码

break;

case ARGS: //此处仅处理各实参表达式的求值的代码序列，不生成ARG的实参系列

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

Exp(T->ptr[0]);

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

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

if (T->ptr[1]) {

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

Exp(T->ptr[1]);

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

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

}

break;

}

}

}

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;

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

T->type=T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT;

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

T->ptr[1]->width=T->type==INT?4:8; //将一个变量的宽度向下传递

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

T->width=(T->type==INT?4:8)\* 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

T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT;//获取函数返回类型送到含函数名、参数的结点

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在退出一个复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;

T->ptr[1]->type=!strcmp(T->ptr[0]->type\_id,"int")?INT:FLOAT; //确定变量序列各变量类型

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;

width=T->ptr[1]->type==INT?4:8; //一个变量宽度

while (T0) { //处理所以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());

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));

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]);

/\*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 ID:

case INT:

case FLOAT:

case ASSIGNOP:

case AND:

case OR:

case RELOP:

case PLUS:

case MINUS:

case STAR:

case DIV:

case NOT:

case UMINUS:

case FUNC\_CALL:

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);

}

# 参考文献

[1] Jobn Levine著 陆军 译. 《Flex与Bison》.东南大学出版社

[2] 许畅等编著. 《编译原理实践与指导教程》.机械工业出版社

[3] 王生原等编著. 《编译原理（第3版）》.清华大学出版社