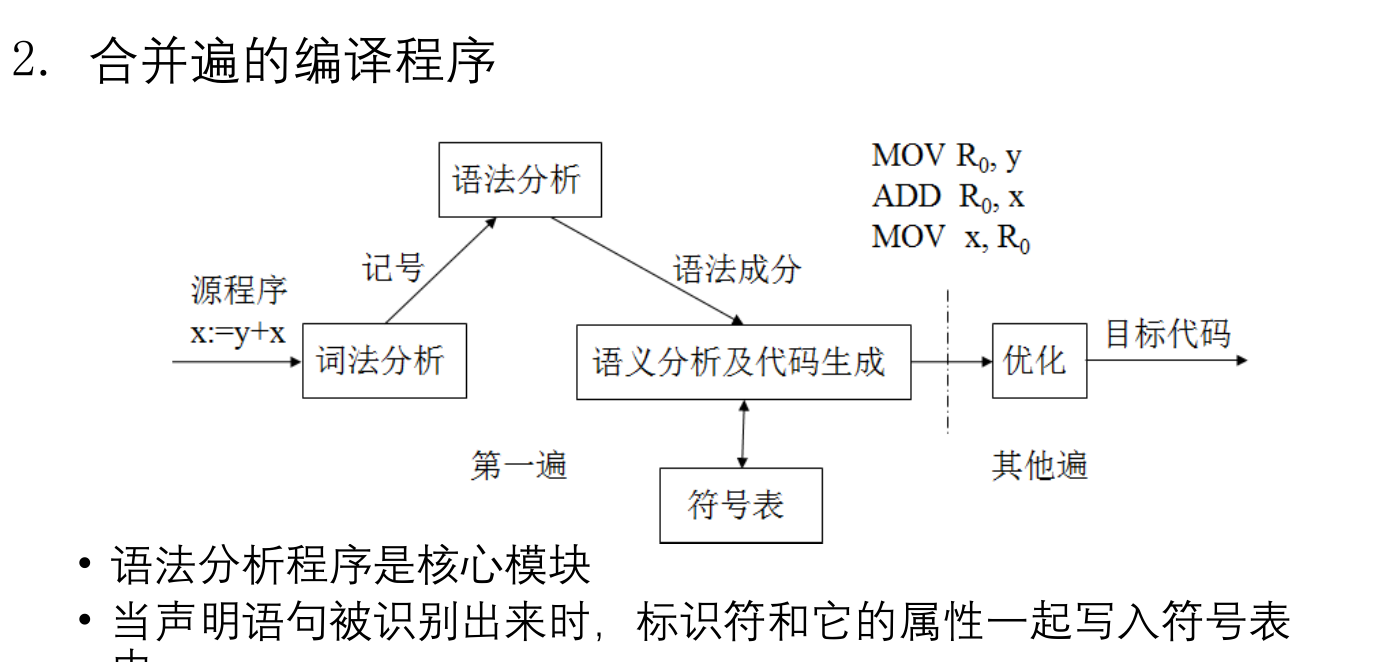
## 0 程序架构

### 架构

由于采用flex-bison，主体时语法分析程序，故采用合并遍的结构



### 各阶段输入输出

## 1 语言说明

### 1.1 顶层定义top

Program -> ExtDefList //全局定义

ExtDefList -> ExtDef ExtDefList

ExtDef -> VarDec

| FuncDef

VarDec -> Specificer VarList ; //变量声明

| Specificer ;

VarList -> Var, VarList | Var

Var -> ID | ID = Exp | Var [ C\_INT ]

FuncDef -> Specificer FuncHeader CompSt //函数定义

FuncHeader -> ID(ParamList)

ParamList -> Param, ParamList | Param | ε

Param -> Specificer ID

### 1.2 语句stmt

CompSt -> {StmtList} //复合语句 | 语句块

StmtList -> Stmt StmtList | ε

Stmt -> Exp;

| CompSt

| VarDec

| ID = Exp;

| ID OP\_ASSIGN Exp;

| IF ( Exp ) Stmt

| IF ( Exp ) Stmt ELSE Stmt

| WHILE ( Exp ) Stmt

| RETURN Exp;

### 1.3 表达式Exp

Exp -> ID

| C\_INT

| C\_FLOAT

| C\_CHAR

| TRUE | FALSE

| ( Exp )

| Exp , Exp

| Exp + Exp | Exp – Exp

| Exp \* Exp | Exp / Exp | Exp % Exp

| Exp REL\_OP Exp

| S\_PLUS Exp | S \_MINUS Exp | Exp S \_PLUS | Exp S \_MINUS

| Exp AND Exp

| Exp OR Exp

| ! Exp

| - Exp

| ID ( RParamList )

| ID . ID

RParamList ->RParam, RParamList | RParam | ε //实参列表

RParam -> Exp

### 1.3 具体取值

Specificer -> INT

| FLOAT

| CHAR

| BOOL

| STRUCT TagOpt { VarDecList }

| STRUCT ID

TagOpt -> ID | ε

VarDecList -> VarDec VarDecList | VarDec

## 2 词法分析

### 2.1 记号说明token

由于Bison中定义的记号默认从258开始编码，故单个字符的记号将其默认编码为其ASCII值。

1. 杂项

ID //关键字

C\_INT //int常量

C\_FLOAT //float常量

C\_CHAR //char 常量

1. 操作符//

. //dot

//R\_ARROW //->

S\_PLUS //++

S\_MINUS //--

+, -, \*, /, %

REL\_OP // <, >, <=, >=, ==, !=

AND // &&

OR // ||

!

, //comma

=

OP\_ASSIGN // +=, -=, \*=, /=

1. 边界符

;

()

{}

1. 关键字

INT,

FLOAT,

CHAR,

VOID,

IF,

ELSE,

WHILE,

RETURN,

BOOL,

TRUE,

FALSE,

### 2.2 flex说明

1. “ … ”双引号括起来的字符中除了转义字符之外其它的元字符（含特殊含义的字符如\*代表重复0个或多个）将失去其特殊意义。
2. flex处理冲突方式：1）对于同一个正则表达式，尽可能匹配多的字符。 2）对于不同正则表达式，优先匹配词法分析器中更早定义的模式。
3. 辅助部分定义的正则表达式不能换行，或(|)之间不能有空格
4. 规则部分不能有空行，使用/\*\*/时，要以一个空格开头

## 3 语法分析

### 3.1 非终结符

Program ExtDefList ExtDef VarDec FuncDef Specificer VarList Var FuncHeader ParamList CompSt Param

StmtList Stmt Exp RParamList Rparam

由简到繁地实现

先定义Program -> Exp

然后实现Exp

之后再修改Program定义，依次实现Stmt, CompSt, VarDec, FuncDef等。

### 3.2 bison说明

1 编译参数-d 生成xxx.tab.h。包含终结符（记号）的宏定义（或枚举定义）

2 编译参数-v 生成parser.output。说明（LALR（1）分析法）哪几个状态有多少个冲突项

3 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中通过yylval的成员保存单词属性值，在Bison中就可以通过yylval的成员取出属性值，实现了数据的传递。

4 冲突处理

Bison使用LALR分析法，会自动报告冲突。类似于Exp->Exp + Exp | Exp \* Exp的文法易于理解，但会产生二义性。在产生冲突后，可以通过指定结合性，和优先级来解决冲突。

%left %right %noassoc（没有结合性）%prec（对产生式指定优先级，默认为规则最右边记号的优先级）

5 Not found unistd.h错误：在windows下没有unistd头文件，使用VS编译会报错。（MinGW中有该头文件，编译不报错） 解决办法：create a dummy header file named unistd.h, include io.h and process.h

6 How to run it? 在make编译生成.exe文件后，直接运行，输入一些字符后输入EOF (windows下为ctrl-Z, linux下为ctrl-D)。或者使用重定向，’<’将文件内容作为输入，’>file 2>&2’将输出和错误输出到文件里。即：my\_gcc.exe <sample.c >output 2>&1

或者编写一个main函数文件，将yyin变量extern声明一下。然后再main函数中将yyin赋值为一个文件的FILE \*指针。之后词法分析器便会从该文件中读取输入，因此语法分析器通过调用yylex读取到的记号便是该文件的了。

7 yylloc，在.y文件中声明%locations选项，生成的parse.tab.h中便会包含YYLTYPE定义（一个结构体，用于存储行号，列号），在.l文件中定义yylloc变量; 之后便可以使用该变量传递位置信息给语法分析器。

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

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

YY\_USER\_ACTION宏会在.l文件中识别正则表达式后，执行对应代码之前调用

### 3.3 union

在.y文件的第一部分可以通过%union定义一个联合体类型。该联合体成员变量可用于说明终结符和非终结符的类型。

bision -d编译.y文件生成的xxx.tab.h中包含union YYSTYPE的定义，并声明了从其它文件导入yylval变量，因此我们需要在lex.l中定义yylval

typedef *union* YYSTYPE

{

/\*\*/

} YYSTYPE

extern YYSTYPE yylval;

通过yylval，我们可以在词法分析器和语法分析器中传递数据。

比如REL\_OP为终结符，可能的属性值为”==” “<”等。

我们在.l文件中，识别到对应记号时便将yylval.type\_relop存储对应的值，从而在.y文件规约Exp REL\_OP Exp时，生成对应名字的中间节点。

在.y文件中$X会被翻译成 yylval.<终结符或非终结符的类型>

### 3.3 AST的表示

AST不同于语法分析树，忽略了一些用于表示层次关系的节点。

#### 孩子表示法

每个节点有固定个若干个子节点代表child

优点：实现简单

缺点：空间开销大

#### 孩子-兄弟表示法

每个节点有其第一个孩子和其右边第一个兄弟节点的指针（二叉链表）

优点：存储效率高

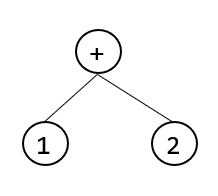
缺点：遍历时要复杂些。

#### 叶子节点和内部节点

Exp -> Exp + Exp

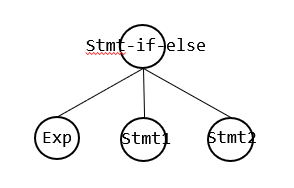
Exp -> C\_INT

对于输入1 + 2，AST为



Stmt -> IF(Exp) Stmt1 ELSE Stmt2

可生成一个内部节点



叶子节点：（一些终结符）

C\_INT, C\_FLOAT, C\_CHAR, //数值常量

TRUE, FALSE //关键字，相当于布尔型常量

INT, FLOAT, CHAR, BOOL, //类型声明

内部节点：（剩余终结符+所有非终结符）

#### 节点定义

采用孩子兄弟表示法。

内部节点定义：（除去了行号等一些变量）

struct node{

char name[20]; //标识节点名称，如Exp+Exp可产生一个名为"+"的内部节点

struct node \*child, \*sibling;

}

叶子节点定义：

struct leaf{

struct node \*child, \*sibling;

int leaf\_type; //用于标识采用下面联合体的哪个属性，true, false节点特殊，leaf\_type直接可以表示其值。 可以复用终结符（记号）的编码

union{

int type\_int; //用于整数叶子节点

float type\_float;

char type\_char;

char type\_id[20];

}

}

### 3.4 冲突解决

#### 算数表达式冲突

采用类似Exp -> Exp + Exp | Exp \* Exp的文法书写十分简便，但会导致二义性。如1 + 2 \* 3 可产生两种不同的语法分析树。通过定义优先级和结合性可以解决冲突：

%left ','

%right '=' OP\_ASSIGN

%left OR

%left AND

%left REL\_OP

%left '+' '-'

%left '\*' '/' '%'

%right S\_PLUS S\_MINUS '!'

其中:

1. 定义在下面的优先级更高；
2. %left和%right分别代表左结合性，和右结合性。

#### If-than和if-else冲突

文法

Stmt -> if ( Exp ) Stmt

Stmt -> if ( Exp ) Stmt else Stmt

对于if(e1) if(e2) else s有两种语法分析树

解决：

%nonassoc IF

%nonassoc ELSE

Stmt:

    IF '(' Exp ')' Stmt             %prec IF

    | IF '(' Exp ')' Stmt ELSE Stmt %prec ELSE

#### 函数调用实参和逗号运算符冲突

由于Exp -> Exp, Exp

而Exp -> ID ( RParamList ) ;

RParamList -> RParam, RParam | RParam | ε

RParam -> Exp

即Exp, Exp可看成 Exp -> Exp, Exp

也可看成 RParam, Par RParam am

RParam -> Exp

解决：

RParam:

    Exp                         %prec ','

;

## 4 语义分析

### 4.1 语义分析任务

主要任务便是确认程序的语义是正确的，具体任务有：

1 管理符号表

2 声明符号时，进行重定义检查（变量定义，函数定义）

3 引用符号时，进行未定义检查和作用域检查（一起进行）（引用变量，函数调用）

4 处理表达式时，进行类型检查。如果是函数调用，则还需进行参数个数检查，参数类型检查。

### 4.2 语义错误类型定义

为了简单起见，变量类型只考虑int和float。

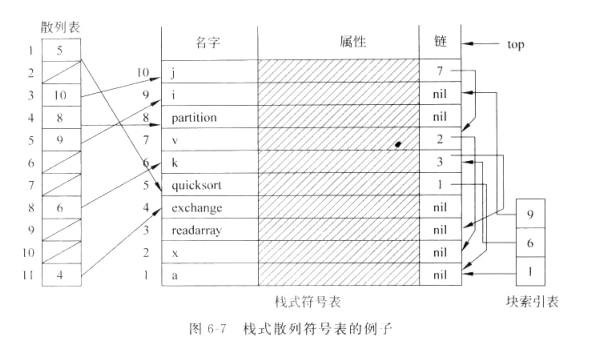
没有对结构体，char，bool进行分析，相应的处理逻辑都留了空白。

也没有考虑函数调用时的实参类型，只考虑了实参的个数。

由于错误7实验进行了要求，因此之后应该会补上bool类型的对应分析。

|  |  |
| --- | --- |
| 错误编号 | 类型 |
| 1 | 变量重定义 |
| 2 | 变量未定义 |
| 3 | 函数重定义 |
| 4 | 函数未定义 |
| 5 | 函数调用时，参数个数错误 |
| ~~6~~ | ~~函数调用时，参数类型错误~~ |
| ~~7~~ | ~~非int, float类型参与算数运算~~ |
| 8 | Int和float进行运算（假设禁止隐式类型转换） |
| 9 | 赋值时，变量类型错误 |
| ~~10~~ | ~~函数定义返回类型和定义不一致~~ |

### 4.3 符号表组织-栈式散列符号表



(《编译原理与技术》，李文生p193)

结构体定义：

typedef *struct*  //符号表

{

    symbol stack[SYMBOL\_TABLE\_MAX\_SIZE];

*int* top;

}SYMBOL\_TABLE;

typedef *struct*  //块索引表

{

*int* stack[BLOCK\_INDEX\_TABLE\_MAX\_SIZE];

*int* top;

}BLOCK\_INDEX\_TABLE;

约定ST(symbol\_table)代表符号表, BIT(block\_index\_table)代表块索引表

需要完成以下几个函数：

int search\_symbol\_table(char \*name);

该函数在访问一个符号时被调用（算数表达式中引用变量，函数调用，赋值语句），由于我们的符号表底层使用的是数组，所以返回一个整型的索引就可以确定符号在符号表中的位置。（因此不必返回一个symbol \*）。当没有找到时，返回-1，调用者根据它打印相关错误信息。

int insert\_symbol\_table(char \*name, 符号的其它属性);

该函数在声明一个符号时被调用（变量声明，函数定义）。如果在当前作用域找到相同的项，则返回-1表示插入失败。否则返回插入位置的索引。由于块索引表栈顶记录了当前作用域的起始位置，因此不必查找索引小于其值的符号表项。

void location();

定位操作，当进入一个新的块时，执行该操作，将当前符号表栈顶索引(ST.top)压入到块索引表栈顶。需要注意的是，对于函数的形参，这里把它当作函数体的一部分，处于同一个作用域中。因此在识别到函数形参时进行定位，进入函数体时不再次进行定位。

Void relocation();

重定位操作，当离开一个块时，执行该函数，将当前符号表栈顶索引(ST.top)恢复为块索引表栈顶的值，且块索引表弹栈。相当于将符号表恢复到进入函数之前。

### 4.4 遍历AST进行语义分析

采递归的方式

## 5 中间代码生成

IR (Intermediate Representation) 在前后端分离的编译器中起着承上启下的作用。

### 5.1 中间语言IR Language

采用三地址代码TAC作为中间语言，并采用四元式实现。

TAC定义如下表所示。

**表5-1 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 |  | 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 | MULT | 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 | Z |
| RETURN x | 返回语句 | RETURN |  |  | X |
| ARG x | 传实参x | ARG |  |  | X |
| x:=CALL f | 调用函数 | CALL | F |  | X |
| PARAM x | 函数形参 | PARAM |  |  | X |
| READ x | 读入 | READ |  |  | X |
| WRITE x | 打印 | WRITE |  |  | X |

s

### 5.2 由简入繁

1 赋值语句

S -> ID = Exp

Exp -> Exp op Exp

Exp -> ID

Exp -> C\_IN