

华中科技大学网络安全学院

# 编译原理实验指导教程

2019 年 10 月

编译原理课程组

本实验指导教程作为编译原理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，第一个需要完成的任务：1) 定义一个待实现编译器的语言，用上下文无关文法定义该语言，2) 写出 10 个该语言编写的程序样例（覆盖所有文法规则），用于后续测试，3) 给该语言起一个名称。（任务一）

后续的工作就是完成该语言的编译器。

以下给出的是一个简化的 C 语言的文法作为参考例子，将其称为 mini-c。

G[program]:

program  $\rightarrow$  ExtDefList

ExtDefList  $\rightarrow$  ExtDef ExtDefList  $\mid \epsilon$

ExtDef  $\rightarrow$  Specifier ExtDecList ; | Specifier FunDec CompSt

Specifier  $\rightarrow$  int | float

ExtDecList  $\rightarrow$  VarDec | VarDec , ExtDecList

VarDec  $\rightarrow$  ID

FucDec  $\rightarrow$  ID ( VarList ) | ID ( )

VarList  $\rightarrow$  ParamDec , VarList | ParamDec

ParamDec  $\rightarrow$  Specifier VarDec

CompSt  $\rightarrow$  { DefList StmList }

StmList  $\rightarrow$  Stmt StmList  $\mid \epsilon$

Stmt  $\rightarrow$  Exp ; | CompSt | return Exp ;

| if ( Exp ) Stmt | if ( Exp ) Stmt else Stmt | while ( Exp ) Stmt

DefList  $\rightarrow$  Def DefList |  $\epsilon$

Def  $\rightarrow$  Specifier DecList ;

DecList  $\rightarrow$  Dec | Dec , DecList

Dec  $\rightarrow$  VarDec | VarDec = Exp

Exp  $\rightarrow$  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 ( )

Args  $\rightarrow$  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。实验时，要求自己重新定义符合实验要求的语言的文法。

在后续的章节里，以语言 mini-c 为例，提供编译程序构造各阶段实现的指导。



## 2 词法分析与语法分析

第一个实验的第二项任务：1) 构建词法、语法分析器，2) 用测试样例覆盖所有文法规则，3) 能检出 10 个不同的词法错误，4) 具有错误恢复功能，能检出 10 个不同的语法错误，并提示行号 5) 对正确的样例，输出其语法树（任务二）。

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。

其中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，其中通过调用 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，其中调用 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，作为语法规则的终结符，同时用 Bison 编译后，生成一个文件 Parser.tab.h，该文件中将这些标识符作为枚举常量，每一个就对应一个（类）单词。这些枚举常量提供给 Lex.l 使用，每一个对应一个单词的种类码；同时在 Parser.y 中，这些枚举常量，每一个对应一个终结符。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。

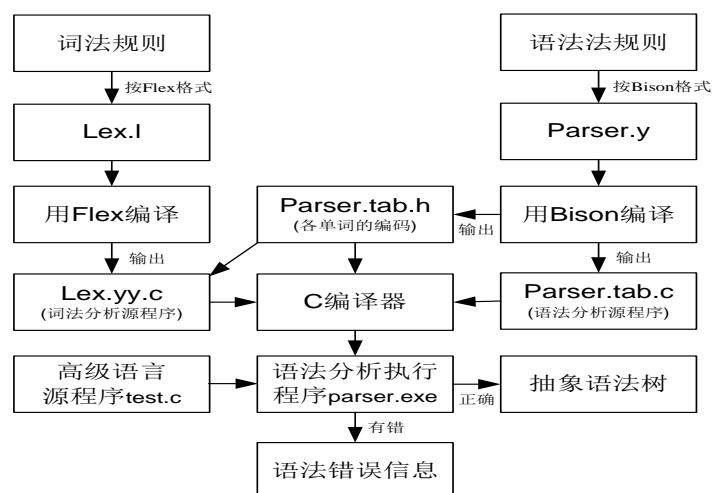


图 2-1 使用 Flex 和 Bison 构建语法分析器工作流程

这个流程中，以语法分析作为主体，在语法分析过程中，每当需要读入下一个符号（单词）时，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。

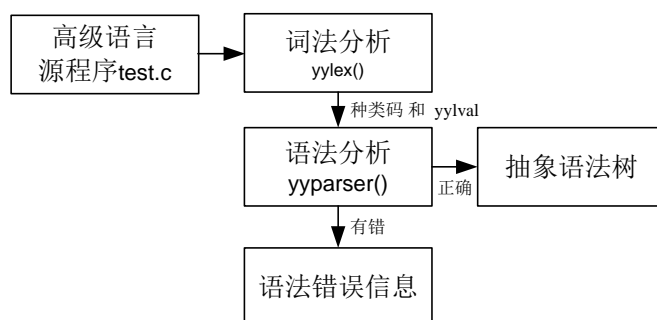


图 2-2 词法、语法分析器控制流程

## 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规表达式。用正规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，该工具要求以正规表达式（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规表达式就是关键。

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。

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规式宏名的定义，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：

#### 正规表达式      动作

表示词法分析器一旦识别出正规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，作为实验检查的依据。](#)

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。

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。

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。通常语义分析是采用语法制导的语义分析，所以在语法分析的同时还可以完成部分语义分析的工作，在



Bison 文件中还会包含一些语义分析的工作。Bison 程序的扩展名为 .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

其中：%{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

在实验中要用到的几个主要内容有：

(1) 语义值的类型定义，mini-c 的文法中，有终结符，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，值得类型会有些差异，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（结构类型）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 {  
    .....  
}
```

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

(2) 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?这时需要做的是[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。

(3) 非终结符的属性值类型说明,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,这个类型来源于(1)中联合的某个成员,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 ExtDefList 属性值的类型对应联合中成员 ptr 的类型,在本实验中对应该一个树结点的指针。

(4) 优先级与结合性定义。对 Bison 文件进行翻译,得到语法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,这是,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的设定解决,例如对表达式 Exp,其部分文法规则有:

$$\text{Exp} \rightarrow \text{Exp} + \text{Exp} \quad | \text{Exp} - \text{Exp} \quad | \text{Exp} * \text{Exp} \quad | \text{Exp} / \text{Exp}$$

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，前面符号的优先级低。

另外就是对： $\text{Exp} \rightarrow -\text{Exp}$  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-，种类码都是 MINUS，为此，需要在定义一个优先级搞得单目-符号 UMINUS：

```
%left '+' '-'
%left '*' '/'
%right UMINUS
```

相应对其规则处理为：

```
 $\text{Exp} \rightarrow -\text{Exp}$  %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，要去掉全部的冲突，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:  $\text{Exp} \rightarrow \text{Exp} = \text{Exp}$ , 在 `parser.y` 中为:

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

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,第二棵对应是 `$3` 的表示的右值表达式的子树,另外 `yylineno` 表示行号。

通过上述给出的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 2.3 详细叙述。

在使用 Bison 的过程中,要完成**报错**和**容错**,使用 Bison 得到的语法分析程序,对 `mini-c` 程序进行编译时,一旦有语法错误,需要准确、及时的报错,这个由 `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、哪一列有错,这个可参见参考文献[2]。

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,一次尽可能多的报出源程序的语法错误,减少交互次数,提高编译效率。这时课通过跳过指定的符号,例如:

```
Stm →error SEMI
```

表示对语句分析时,一旦有错,跳过分号 (`SEMI`),继续进行语法分析。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]。

## 2.3 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，去掉了一些修饰性的单词部分，简明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。

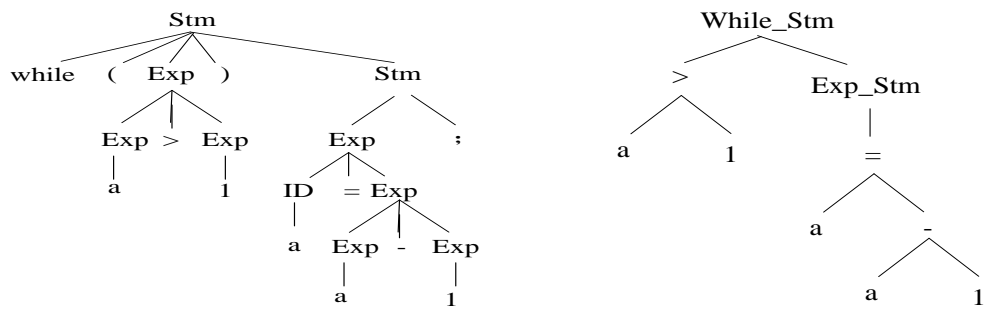


图2-3 推导树和抽象语法树的形式

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。但从形式上看，AST 简洁很多，这样后续处理相应就方便得多。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	可忽略	

由上表可见，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;

```

这样所有符号的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 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。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，标识符的字符串值保存在成员 `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，所有结点的类型也是统一的，所以为区分结点的属性，在定义结点时，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，下面就需要考虑其物理结构，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 3 个指针域，可指向 3 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。

在 `parser.y` 中，在完成归约的过程中，完成抽象语法树的构造。例如处理

下面规则，即完成将 INT 归约成 Exp 时。

**Exp: INT**

需要调用函数 `mknnode` 生成一个类型为 INT 的叶结点，指针赋值给 Exp 的属性 `$$` (`$$=mknnode(INT,NULL,NULL,NULL,yylineno)`)；，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 (`$$->type_int=$1;`)。

当处理下面规则，即完成将 `Exp1+Exp2` 归约成 Exp 时。

**Exp: Exp1 PLUS Exp2**

需要调用函数 `mknnode` 生成一个类型为 PLUS 的非叶子结点，结点指针赋值给 Exp 的属性 `$$` (`$$=mknnode(PLUS,$1,$3,NULL,yylineno)`)。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。编译命令格式：`gcc -o parser ast.c parser.tab.c lex.yy.c`。例如将图 2-4 所示的 mini-c 程序 `test.c` 作为测试程序。

```
1  int a,b,c;
2  float m,n;
3  int fibo(int a) {
4      if (a==1 || a==2) return 1;
5      return fibo(a-1)+fibo(a-2);
6  }
7  int main() {
8      int m,n,i;
9      m=read();
10     i=1;
11     while (i<=m)
12     {
13         n=fibo(i);
14         write(n);
15         i=i+1;
16     }
17     return 1;
18 }
19
```

图2-4 AST生成测试程序

该程序对应的 AST 显示如下。可以看到显示结果重点突出了程序的语法结构，去掉了分隔符等信息，能很方便地由这个显示结果还原出上述程序代码。

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**

通过对 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。

## 2.4 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 文件即可,然后安装。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会影响以后的使用。可如此: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 的主页下载安装。

下载地址:

<http://sourceforge.net/projects/mingw/files/latest/download?source=files>

安装过程中,会自动开启控制台,我们仅需稍等片刻,任其自动完成。安装完毕后,将 c:\gnuwin32\lib 里面的 libfl.a 和 liby.a 复制到 C:\MinGW\lib 里面。现在该安装的都已安装完毕,那么我们该设置环境变量了。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,修改,在后面加上 c:\gnuwin32\bin 和 C:\MinGW\bin (也可以使用其它的 C 编译,如 codeblocks)。注意每一个路径是用分号分隔的,然后写第一个路径,然后分号,第二个路径。

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。

## 3 语义分析

前两周的实验，完成了语言的定义及实验一的两个子任务，能对源程序样例进行词法和语法分析，报错，对正确的源程序，建立了抽象语法树 AST。在此基础上，进行第二个实验。

**第二个实验任务：**1) 用不同的样例覆盖语法规则，进行语义分析，遍历对应的 AST 树，发现语义的错误并输出到屏幕 2) 建立符号表，并在合适的分析位置上输出符号表。

(下一章是第三次的实验，是再次遍历 AST 树，完成中间代码生成。)

### 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，于是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

**任务：**语义分析这部分完成的是静态语义分析检查，请你完成包括以下功能：

(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，那么久就出现一个语义错误。另外，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 语义程序参考架构

前面的实验中，完成了 AST 的生成，现在可以通过遍历 AST，完成符号表的管理和静态语义分析检查等任务。

#### 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。

举例：对于一个局部变量说明语句：int a, b；实验一中，已经进行词法语法分析，产生该说明语句其对应的局部 AST 树结构（通过二叉树保存），示意图如图 3-1 所示。

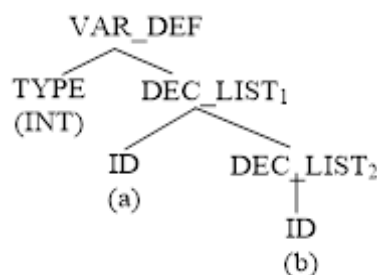


图 3-1 说明语句的 AST 形式

当对 AST 树进行遍历的过程中，1) 第一次访问到 VAR\_DEF 结点，然后按遍历访问次序，2) 接着访问 TYPE 结点，在确定 TYPE 结点的数据类型为 INT 之后，3) 将回到 VAR\_DEF，此时，该说明语句的变量列表成分中，每个变量的类型实际上已经可以确定了，都是 INT 类型，所以，可将此类型属性，向下传给结点 DEC\_LIST<sub>1</sub>。接着类型由 DEC\_LIST<sub>1</sub> 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\_LIST<sub>1</sub> 传到 DEC\_LIST<sub>2</sub> 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，根据不同语法成分在语义翻译中的需要，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，并且一个作用域内不允许名字重复。为此，可以通过一个全局变量 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位 1；处理函数形

式参数时，固定形参名在填写符号表时，层号也为 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) 符号表的每次遇到一个复合语句的结点 COM\_STM，首先对 LEV 加上 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，检查层号为 LEV 的符号是否和当前待登记的符号重名，如果发现了重名定义，则输出重复定义错误信息；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的结点 COM\_STM 的子树，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局部变量。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，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，当分析各种表达式，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，没有找到该符号，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，会按就近优先的原则选中符号。当查找到符号后，可以做进一步的进行语义分析，如：(1) 当进行函数调用时，如果根据函数名在符号表找到匹配的符号，但符号类型是变量而不是函数，则需要输出报错；(2) 当函数调用时，根据函数名找到这个函数，需要判断参数个数、类型与调用是否匹配；(3) 当需要变量，而根据变量名查找到符号表对应类型是函数，也需要输出报错。类似的还有其他检查，都需要逐一进行，逐项输出错误提示信息。

对图 2-4 中的测试程序样例，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，其中，read 和 write 为预先定义好的函数，可以直接使用；表中的第 4 行（变量名 a 别名 v1...），对应源程序图 2-4 中的 int a 所在行。

变量名	别名	层号	类型	标记
read		0	int	F
x		1	int	P
write		0	int	F
a	v1	0	int	V
b	v2	0	int	V
c	v3	0	int	V
m	v4	0	float	V
n	v5	0	float	V
fibo	v6	0	int	F
a	v7	1	int	P

图 3-2 符号表内容一

接着，函数 fibo 的函数体对应复合语句时分析完成后，需要删除 fibo 中的局部变量(本例子中函数 fibo 没有局部变量)，然后开始访问 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，此时的符号表中的内容输出，为如图 3-3 所示。

变量名	别名	层号	类型	标记
read		0	int	F
x		1	int	P
write		0	int	F
a	v1	0	int	V
b	v2	0	int	V
c	v3	0	int	V
m	v4	0	float	V
n	v5	0	float	V
fibo	v6	0	int	F
a	v7	1	int	P
main	v8	0	int	F
m	v9	1	int	V
n	v10	1	int	V

图 3-3 符号表内容二

在语义分析编码过程中，应可以在任意点输出并显示符号表的内容，观察是否正确地管理好了符号的作用域。实验中可以在一段源程序之后，输出对应符号表，包括全局变量定义，函数、符合语句之后。



## 4 中间代码的生成

通过前面对 AST 遍历，完成了语义分析后，没有语法、语义错误时，对 AST 进行遍历，计算相关的属性值，已经可以建立并随时访问符号表，那么接下来，可以在遍历 AST 树时，增加中间代码生成所需要的功能，要求生成以三地址代码格式 TAC 作为中间语言的中间语言代码序列。**实验任务：按前面构造的不同的样例为输入（覆盖语法结构的不同分支），输出其对应的 TAC 序列。**

为了便于实现，在这里对本实验做了一些限制：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。对学有余力的同学，其它数据类型的实现，可自行选择。

### 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		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
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
---------	----	-------	--	--	---

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。

因下次实验，还需要根据这里产生的 TAC 序列，进一步翻译生成目标代码，所以设计 TAC 序列的存储结构时，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：这些部分包含的数据类型有多种，整常量，实常量，还有使用标识符的情况，如变量的别名、变量在其数据区的偏移量和层号、转移语句中的标号等。类型不同，所以可以考虑使用联合数据类型进行存储。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 信息可以说明联合中的有效成员是整常量、实常量、标识符表示的别名、标号或函数名等。

(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，可以用双向循环链表存放 TAC 代码序列。

## 4.2 翻译模式

参考文献[3]中的 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。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  $A \rightarrow M \dots X \dots N$  的翻译模式：

$A \rightarrow M \dots \{X \text{ 的继承属性计算}\} X \dots N \quad \{A \text{ 的综合属性计算}\}$

对应的 AST 中的形式如图 4-1 所示。

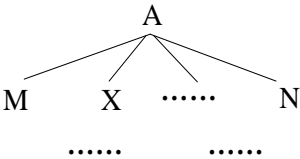


图 4-1 翻译模式规则部分的 AST

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 之前的结点的属性，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；在翻译模式中规则的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的计算，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，以及每次完成了计算后，中间结果需要用临时变量保存，临时变量也需要登记到符号表中。另外由于使用复合语句，可以使作用域嵌套，不同的作用域中的变量允许同名（这是在 mini-c 中，和 C 语言一样采用就近优先的原则），但是，在 TAC 中间语言中，因为已经没有复合语句结构来区分不同的层次，所以在 TAC 中需要区分同名变量（换名）。方法：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，实现符号名数据的唯一性。翻译时，要将对变量的操作，替换成对相应别名的操作，别名命名格式，可以参考为 **v+序号**（v1, v2, ...）；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标记（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，可以利用这个属性，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，方便计算变量、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。如果采用链表存放中间代码序列，该属性就是指向该节点对应中间代码链表的头指针。

**.Etrue (.Efalse)** 在完成布尔表达式翻译时，表达式值为‘真’（或为‘假’）时，要转移的程序位置（可以用标号的字符串形式表示）。

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

为了生成中间代码序列，定义了几个函数：

**newTemp** 生成一个临时变量，并登记到符号表中，以 temp+序号的形式组成的符号串作为临时变量的别名，符号名称栏可以用空串，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 2 个运算对象和 1 个运算结果。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，方便最后的目标代码生成。全局变量也需要带上偏移量。

**genLabel** 生成标号语句。

以上 3 个函数，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。过程可参考课件 if\_then\_else 语句的翻译过程。

对图 2-4 中的测试程序，需要你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
  temp14 := CALL write
  temp15 := #1
  temp16 := v11 + temp15
  v11 := temp16
  GOTO label10
LABEL label8 :
  temp17 := #1
  RETURN temp17
LABEL label5 :
```

## 5 目标代码的生成

**任务：**将 TAC 的指令序列转换成目标代码。同前一章一样，对实现做了一些简化，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 MIPS 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，完成一个可运行的编译程序。

### 5.1 目标语言的指令定义

目标语言可选定 MIPS32 指令序列，可以在 SPIM Simulator 上运行，SPIM Simulator 的安装使用参见文献[2]。TAC 指令和 MIPS32 指令的对应关系，如表 5-1 所示。其中  $\text{reg}(x)$  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 $\text{reg}(x), k$
$x := y$	move $\text{reg}(x), \text{reg}(y)$
$x := y + z$	add $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y - z$	sub $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y * z$	mul $\text{reg}(x), \text{reg}(y), \text{reg}(z)$
$x := y / z$	div $\text{reg}(y), \text{reg}(z)$ mflo $\text{reg}(x)$
GOTO $x$	j $x$
RETURN $x$	move $\$v0, \text{reg}(x)$ jr $\$ra$
IF $x == y$ GOTO $z$	beq $\text{reg}(x), \text{reg}(y), z$
IF $x != y$ GOTO $z$	bne $\text{reg}(x), \text{reg}(y), z$
IF $x > y$ GOTO $z$	bgt $\text{reg}(x), \text{reg}(y), z$
IF $x \geq y$ GOTO $z$	bge $\text{reg}(x), \text{reg}(y), z$
IF $x < y$ GOTO $z$	ble $\text{reg}(x), \text{reg}(y), z$
IF $x \leq y$ GOTO $z$	blt $\text{reg}(x), \text{reg}(y), z$

X:=CALL f	jal f move reg(x),\$v0
-----------	---------------------------

## 5.2 寄存器的分配

有关寄存器等详细描述参见文献[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，数据流的分析介绍很少，这样实现起来会有较大难度。

在实验中，不对寄存器的分配算法做任何具体要求。**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) ble \$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) 首先根据保存在函数调用指令中的 offset，找到符号表中的函数定义点，获取函数的参数个数 i，这样就可得到在 X:=CALL f 之前的 i 个 ARG 形式的中间代码，获得 i 个实参值所存放的单元，取出后送到形式参数的单元中。再活动记录的空间。

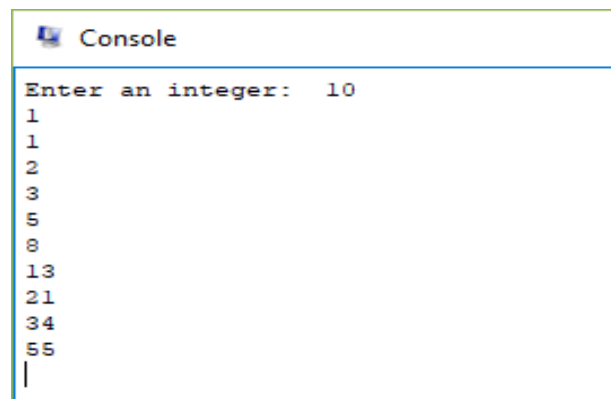
(2) 根据符号表记录的活动记录大小，开辟活动记录空间和保存返回地址。  
思考一下 main 函数的活动记录如何处理？

(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，运行结果如图 5-1 所示。



```
Console
Enter an integer: 10
1
1
2
3
5
8
13
21
34
55
|
```

图 5-1 测试程序运行结果

# 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+yyldeng-1; yycolumn+=yyldeng;
typedef union {
    int type_int;
    int 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表示符的规则，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;
void yyerror(const char* fmt, ...);
void display(struct node *,int);
%}

%union {
    int    type_int;
    float  type_float;
    char   type_id[32];
    struct node *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
Stmnt StmList DefList Def DecList Dec Exp Args

//% 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，由词法分析得到的数值
%token <type_id> ID RELOP TYPE      //指定ID,RELOP 的语义值是type_id，由词法分析得到的标识符字符串
%token <type_float> FLOAT           //指定ID的语义值是type_id，由词法分析得到的标识符字符串

%token LP RP LC RC SEMI COMMA      //用bison对该文件编译时，带参数-d，生成的exp.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

%left ASSIGNOP
%left OR
%left AND
%left RELOP
%left PLUS MINUS
%left STAR DIV
```

```

%right UMINUS NOT

%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(EXT_DEF_LIST,$1,$2,NULL,yylineno);}    //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    {$$=mknode(EXT_VAR_DEF,$1,$2,NULL,yylineno);}    //该结点对
应一个外部变量声明
    | Specifier FuncDec CompSt    {$$=mknode(FUNC_DEF,$1,$2,$3,yylineno);}    //该结点对
应一个函数定义
    | error SEMI    {$$=NULL; }
    ;
Specifier:  TYPE
{$$=mknode(TYPE,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);$ $->type=!strcmp($1,"int"?INT:FLO
AT);}
    ;
ExtDecList:  VarDec    {$$=$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 {$$=mknode(EXT_DEC_LIST,$1,$3,NULL,yylineno);}
    ;
VarDec:  ID    {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}    //ID结
点，标识符字符串存放结点的type_id
    ;
FuncDec: ID LP VarList RP    {$$=mknode(FUNC_DEC,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//
函数名存放在$$->type_id
    | ID LP RP    {$$=mknode(FUNC_DEC,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}//
函数名存放在$$->type_id
    ;
VarList: ParamDec    {$$=mknode(PARAM_LIST,$1,NULL,NULL,yylineno);}
    | ParamDec COMMA VarList    {$$=mknode(PARAM_LIST,$1,$3,NULL,yylineno);}
    ;
ParamDec: Specifier VarDec    {$$=mknode(PARAM_DEC,$1,$2,NULL,yylineno);}
    ;
CompSt: LC DefList StmList RC    {$$=mknode(COMP_STM,$2,$3,NULL,yylineno);}
    ;

```

```

StmList: {$$=NULL; }
        | Stmt StmList {$$=mknode(STM_LIST,$1,$2,NULL,yylineno);}
        ;

Stmt:   Exp SEMI      {$$=mknode(EXP_STMT,$1,NULL,NULL,yylineno);}
        | CompSt      {$$=$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 {$$=mknode(RETURN,$2,NULL,NULL,yylineno);}
        | IF LP Exp RP Stmt %prec LOWER_THEN_ELSE {$$=mknode(IF_THEN,$3,$5,NULL,yylineno);}
        | IF LP Exp RP Stmt ELSE Stmt {$$=mknode(IF_THEN_ELSE,$3,$5,$7,yylineno);}
        | WHILE LP Exp RP Stmt {$$=mknode(WHILE,$3,$5,NULL,yylineno);}
        ;

DefList: {$$=NULL; }
        | Def DefList {$$=mknode(DEF_LIST,$1,$2,NULL,yylineno);}
        ;

Def:   Specifier DecList SEMI {$$=mknode(VAR_DEF,$1,$2,NULL,yylineno);}
        ;

DecList: Dec {$$=mknode(DEC_LIST,$1,NULL,NULL,yylineno);}
        | Dec COMMA DecList {$$=mknode(DEC_LIST,$1,$3,NULL,yylineno);}
        ;

Dec:   VarDec {$$=$1;}
        | VarDec ASSIGNOP Exp
        {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");}
        ;

Exp:   Exp ASSIGNOP Exp
        {$$=mknode(ASSIGNOP,$1,$3,NULL,yylineno);strcpy($$->type_id,"ASSIGNOP");} //$$结点type_id空置未
        用，正好存放运算符
        | Exp AND Exp {$$=mknode(AND,$1,$3,NULL,yylineno);strcpy($$->type_id,"AND");}
        | Exp OR Exp {$$=mknode(OR,$1,$3,NULL,yylineno);strcpy($$->type_id,"OR");}
        | Exp RELOP Exp {$$=mknode(RELOP,$1,$3,NULL,yylineno);strcpy($$->type_id,$2);} //词法分析关
        系运算符符号自身值保存在$2中
        | Exp PLUS Exp {$$=mknode(PLUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"PLUS");}
        | Exp MINUS Exp {$$=mknode(MINUS,$1,$3,NULL,yylineno);strcpy($$->type_id,"MINUS");}
        | Exp STAR Exp {$$=mknode(STAR,$1,$3,NULL,yylineno);strcpy($$->type_id,"STAR");}
        | Exp DIV Exp {$$=mknode(DIV,$1,$3,NULL,yylineno);strcpy($$->type_id,"DIV");}
        | LP Exp RP {$$=$2;}
        | MINUS Exp %prec UMINUS
        {$$=mknode(UMINUS,$2,NULL,NULL,yylineno);strcpy($$->type_id,"UMINUS");}
        | NOT Exp {$$=mknode(NOT,$2,NULL,NULL,yylineno);strcpy($$->type_id,"NOT");}
        | ID LP Args RP {$$=mknode(FUNC_CALL,$3,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | ID LP RP {$$=mknode(FUNC_CALL,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | ID {$$=mknode(ID,NULL,NULL,NULL,yylineno);strcpy($$->type_id,$1);}
        | INT {$$=mknode(INT,NULL,NULL,NULL,yylineno);$->type_int=$1;$->type=INT;}
        | FLOAT
        {$$=mknode(FLOAT,NULL,NULL,NULL,yylineno);$->type_float=$1;$->type=FLOAT;}

```

```

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

%%

int main(int argc, char *argv[]){
    yyin=fopen(argv[1],"r");
    if (!yyin) return;
    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,ylloc.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"

enum node_kind
{ 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,
  FUNC_CALL,ARGS,FUNCTION,PARAM,ARG,CALL,LABEL,GOTO,JLT,JLE,JGT,JGE,EQ,NEQ};
#define MAXLENGTH 1000 //定义符号表的大小
#define DX 3*sizeof(int) //活动记录控制信息需要的单元数
//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
struct opn{
    int kind; //标识操作的类型
    int type; //标识操作数的类型
    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; //变量单元偏移量，或函数在符号表的定义位置序号，目标代码生成时用
};

struct codenode { //三地址TAC代码结点,采用双向循环链表存放中间语言代码
    int op; //TAC代码的运算符种类
    struct opn opn1,opn2,result; //2个操作数和运算结果
    struct codenode *next,*prior;
};

struct node { //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
    enum node_kind kind; //结点类型
    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
    };
    struct node *ptr[3]; //子树指针，由kind确定有多少棵子树
    int level; //层号
```



```

int place;                //表示结点对应的变量或运算结果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];           //该结点对应语句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
char op[10];
int type;                 //结点对应值的类型
int pos;                  //语法单位所在位置行号
int offset;               //偏移量
int width;                //各种数据占用的字节数
};

```

```

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，外部变量名或函数名层号为0，形参名为1，每到1个复合语句层号加1，退出减1
    int type;         //变量类型或函数返回值类型
    int paramnum;     //形式参数个数
    char alias[10];   //别名，为解决嵌套层次使用，使得每一个数据名称唯一
    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
                    //或函数活动记录大小，目标代码生成时使用

    //其它...
};

```

//符号表，是一个顺序栈，index初值为0

```

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，/每到达一个复合语句，将符号表的index值进栈，离开复合语句时，取其退栈值修改符号表的index值，完成删除该复合语句中的所有变量和临时变量\*/

```

    int TX[30];
    int top;
} symbol_scope_TX;

```

```

struct node *mknode(int kind,struct node *first,struct node *second, struct node *third,int pos );
void semantic_Analysis0(struct node *T);
void semantic_Analysis(struct node *T);
void boolExp(struct node *T);
void Exp(struct node *T);
void objectCode(struct codenode *head);

```

## 参考文献

- [1] John 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