第一章 Tiny语言定义

TINY 的程序结构很简单，它在语法上与 Ada 或 Pascal 的语法相似：仅是一个由分号分隔开的语句序列。另外，它既无过程也无声明。所有的变量都是整型变量，通过对其赋值可较轻易地声明变量（类似 FORTRAN 或 BASIC）。它只有两个控制语句： if 语句和 repeat 语句，这两个控制语句本身也可包含语句序列。If 语句有一个可选的 else 部分且必须由关键字 end 结束。除此之外，read 语句和 write 语句完成输入/输出。在花括号中可以有注释，但注释不能嵌套。

TINY 的表达式也局限于布尔表达式和整型算术表达式。布尔表达式由对两个算术表达式的比较组成，该比较使用<与=比较算符。算术表达式可以包括整型常数、变量、参数以及 4 个整型算符+、一、*、/，此外还有一般的数学属性。布尔表达式可能只作为测试出现在控制语句中——而没有布尔型变量、赋值或 I / O。

虽然 TINY 缺少真正程序设计语言所需要的许多特征——过程、数组且浮点值是一些较大的省略——但它足可以用来例证编译器的主要特征了。

一、词法定义

定义记号和它们的特性。TINY 的记号和记号类都列在表 1 中。

TINY 的记号分为 3 个典型类型：保留字、特殊符号和“其他”记号。保留字一共有 8 个，它们的含义类似（尽管直到很后面才需知道它们的语义）。特殊符号共有 1 0 种：分别是 4 种基本的整数运算符号、2 种比较符号（等号和小于），以及括号、分号和赋值符号。除了赋值符号是两个字符的长度之外，其余均为一个字符。

其他记号就是数了，它们是一个或多个数字以及标识符的序列，而标识符又是（为了简便）一个或多个字母的序列。

除了记号之外，TINY 还要遵循以下的词法惯例：注释应放在花括号{ . . . }中，且不可嵌套；代码应是自由格式；空白格由空格、制表位和新行组成；最长子串原则后须接识别记号。

保留字	特殊符号	其他
if	**+**	
then	**-**	**数**
else	*****	**(1 个或更多的数字)**
end	**/**	
repeat	**=<**	
until	**(**	**标识符**
read	**)**	**(1 个或更多的字母)**
write	**;**	
	:=	

表 1

二、语法定义

下面的代码是 TINY 在 BNF 中的文法，我们可从中观察到一些内容： TINY 程序只是一个语句序列，它共有 5 种语句：if 语句、repeat 语句、read 语句、write 语句和 assignment 语句。除了 if 语句使用 end 作为括号关键字（因此在 TINY 中没有悬挂 else 二义性）以及 if 语句和 repeat 语句允许语句序列作为主体之外，它们都具有类似于 Pascal 的语法，所以也就不需要括号或 begin-end 对（而且 begin 甚至在 TINY 中也不是一个保留字）。输入/输出语句由保留字 read 和 write 开始。read 语句一次只读出一个变量，而 write 语句一次只写出一个表达式。

program → *stmt-sequence*
stmt-sequence → *stmt-sequence* **;** *statement* | *statement*
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *exp* **then** *stmt-sequence* **end**
| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *exp*
assign-stmt → **identifier :=** *exp*
read-stmt → **read** *identifier*
write-stmt → **write** *exp*
exp → *simple-exp comparison-op simple-exp* | *simple-exp*
comparison-op → **<** | **=**
simple-exp → *simple-exp addop term* | *term*
addop → **+**|**-**
term → *term mulop factor factor* | *factor*
mulop → ***|/**
factor → **(***exp***)** |**number** |**identifier**

TINY 表达式有两类：在 if 语句和 repeat 语句的测试中使用比较算符=和<的布尔表达式，以及包括标准整型算符+、-、*和/（它代表整型除法，有时也写作 div）的算术表达式（由文法中的 simple-exp 指出）。算术运算是左结合并有通常的优先关系。相反地，比较运算却是非结合的：每个没有括号的表达式只允许一种比较运算。比较运算比其他算术运算的优先权都低。

TINY 中的标识符指的是简单整型变量，它没有诸如数组或记录构成的变量。TINY 中也没有变量声明：它只是通过出现在赋值语句左边来隐式地声明一个变量。另外，它只有一个（全局）作用域，且没有过程或函数（因此也就没有调用）。

还要注意 TINY 的最后一个方面。语句序列必须包括将语句分隔开来的分号，且不能将分号放在语句序列的最后一个语句之后。这是因为 TINY 没有空语句（不同于 Pascal 和 C）。另外，我们将 stmt-sequence 的 BNF 规则也写作一个左递归规则，但却并不真正在意语句序列的结合性，这是因为意图很简单，只需按顺序执行就行了。因此，只要将 stmt-sequence 编写成右递归即可。这个观点也出现在 TINY 程序的语法树结构中，其中的语法序列不是由树而是由列表表示的。

三、语义定义

TINY 语言在其静态语义要求方面特别简单，语义分析程序也将反映这种简单性。在 TINY 中没有明确的说明，也没有命名的常量、数据类型或过程；名字只引用变量。变量在使用时隐含地说明，所有的变量都是整数数据类型。也没有嵌套作用域，因此变量名在整个程序有相同的含义，符号表也不需要保存任何作用域信息。

在 TINY 中类型检查也特别简单。只有两种简单类型：整型和布尔型。仅有的布尔型值是两个整数值的比较的结果。因为没有布尔型操作符或变量，布尔值只出现在 if 或 repeat 语句的测试表达式中，不能作为操作符的操作数或赋值的值。最后，布尔值不能使用 write 语句输出。

四、目标机定义

TINY 语言的代码生成器将生成代码。为了使这成为有意义的工作，我们产生的目标代码可直接用于易于模拟的简单机器这个机器称为 TM (Tiny Machine)。

TINY 语言程序样例：

```
{ sample program
    In TINY language   -
    Computes   factorial
}
read x; { input on integer }
if   0 < x then   { don't compute if x <= 0 }
    fact := 1 ;
repeat
    fact := fact * x;
    x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

第二章 Tiny语言词法分析器设计

从实现方法上来分，本次实验的词法分析器设计分为手编程序代码方式和自动工具 Flex 方式两种。从实现形式上分为控制台方式和图形用户界面方式两种。

一、实验目的

（1）学会针对 DFA 转换图实现相应的高级语言源程序。
（2）深刻领会状态转换图的含义，逐步理解有限自动机。
（3）掌握手工生成词法分析器的方法和 Flex 自动工具词法分析器设计方法，了解词法分析器的内部工作原理。
（4）进一步巩固命令行方式和图形用户界面方式下的程序设计方法；

二、实验内容

1、根据 Tiny 语言的词法定义，编制一个识别 Tiny 语言的词法分析器。从输入的源程序中，识别出各个具有独立意义的记号（Token），即簿记信息（book-keeping tokens）、基本保留字（reserved words）、多字符记号（multicharacter tokens）、特殊符号（special symbols）四大类，如下表所示。并依次输出各个记号的内部编码及记号符号自身值。（遇到错误时可显示“Error”，然后跳过错误部分继续显示）

Reserved Words	Special Symbols	Other	Book-keeping
if	+	number (1 or more digits)	error
then	-	identifier (1 or more letters)	end of file
else	*		
end	/		
repeat	=		
until	<		
read	(		
write	)		
	;		
	:=		

2、从左到右扫描每行该语言源程序的符号，拼成单词，换成统一的内部表示（token）存到符号表。

为了简化程序的编写，有具体的要求如下：
（1）数仅仅是整数。
（2）空白符仅仅是空格、回车符、制表符。
（3）代码是自由格式。
（4）注释应放在花括号之内，并且不允许嵌套

三、实验要求

要求实现编译器的以下功能：
（1）按规则拼单词，并转换成二元式形式
（2）删除注释行
（3）删除空白符 (空格、回车符、制表符)
（4）列表打印源程序，按照源程序的行打印，在每行的前面加上行号，并且打印出每行包含的记号的二元形式（单词种别，单词符号的属性值）
（5）发现并定位错误

词法分析进行具体的要求：
（1）记号的二元式形式中种类采用枚举方法定义；其中保留字和特殊字符是每个都一个种类，标示符自己是一类，数字是一类；单词的属性就是表示的字符串值。
（2）词法分析的具体功能实现是一个函数 getToken ()，每次调用都对剩余的字符串分析得到一个单词或记号识别其种类，收集该记号的符号串属性，当识别一个单词完毕，采用返回值的形式返回符号的种类，同时采用程序变量的形式提供当前识别出记号的属性值。这样配合语法分析程序的分析需要的记号及其属性，生成一个语法树。
（3）标示符和保留字的词法构成相同，为了更好的实现，把语言的保留字建立一个表格存储，这样可以把保留字的识别放在标示符之后，用识别出的标示符对比该表格，如果存在该表格中则是保留字，否则是一般标示符。

四、实验原理

1、Token定义

Symbol	Token	Illustration
if	**IF**	
then	**THEN**	
else	**ELSE**	
end	**END**	
repeat	**REPEAT**	
until	**UNTIL**	
read	**READ**	

write	**WRITE**	
number	**NUM**	
identifier	**ID**	
+	**PLUS**	
-	**MINUS**	
*	**TIMES**	
/		
=	**EQ**	
<	**LT**	
(	**LPAREN**	
)	**RPAREN**	
;	**SEM**	
:=	**ASSIGN**	
error	**ERROR**	
endoffile	**ENDFILE**	

2、数据结构和全局变量

gloable.h 文件中定义：

（1）typedef enum
/* book-keeping tokens */
{ENDFILE, ERROR,
/* reserved words */
IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE,
/* multicharacter tokens */
ID, NUM,
/* special symbols */
ASSIGN, EQ, LT, PLUS, MINUS, TIMES, OVER, LPAREN, RPAREN, SEMI
} TokenType;

（2）几个相关的全局变量：

gloable.h 文件中定义：
extern FILE* source; /* source code text file */
extern FILE* listing; /* listing output text file */
extern int lineno; /* source line number for listing */

scan.h 文件中定义：
/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40
/* tokenString array stores the lexeme of each token */

```
extern char tokenString[MAXTOKENLEN+1];
```

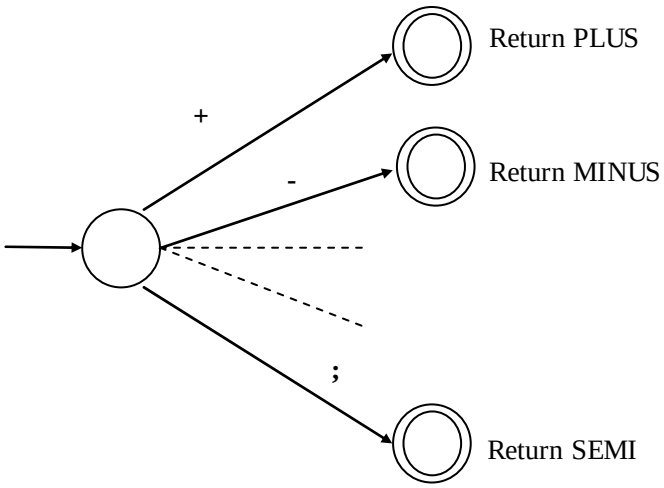
（3）用于观看分析结果的两个全局变量

```
/*************************************************/
/***********      Flags for tracing            ************/
/*************************************************/

/* EchoSource = TRUE causes the source program to
  * be echoed to the listing file with line numbers
  * during parsing */
extern int EchoSource;

/* TraceScan = TRUE causes token information to be
  * printed to the listing file as each token is
  * recognized by the scanner */
extern int TraceScan;
```

3、识别Token的DFA

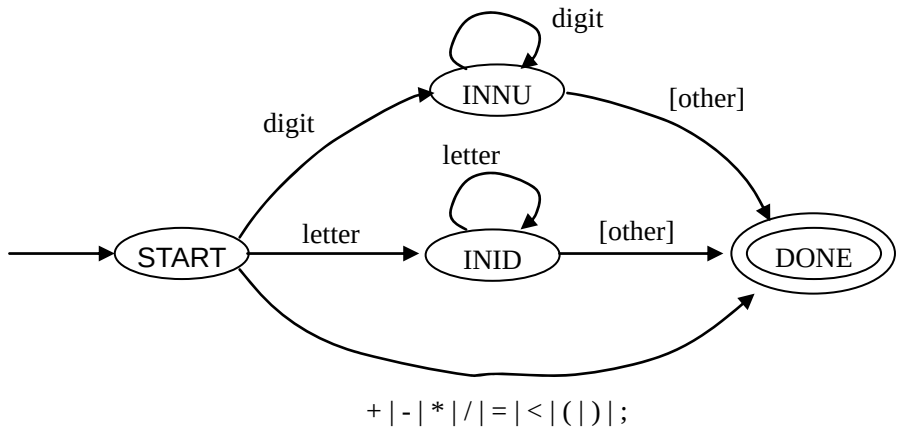
在为该语言设计扫描程序时，可以从正则表达式开始并根据前一节中的算法来开发 NFA 和 DFA。实际上，前面已经给出了数、标识符和注释的正则表达式（TINY 具有更为简单的版本）。其他记号的正则表达式都是固定串，因而均不重要。由于扫描程序的 DFA 记号十分简单，所以无需按照这个例程就可直接开发这个 DFA 了。我们将按以下步骤进行。

首先要注意到除了赋值符号之外，其他所有的特殊符号都只有一个字符，这些符号的 DFA 如下：



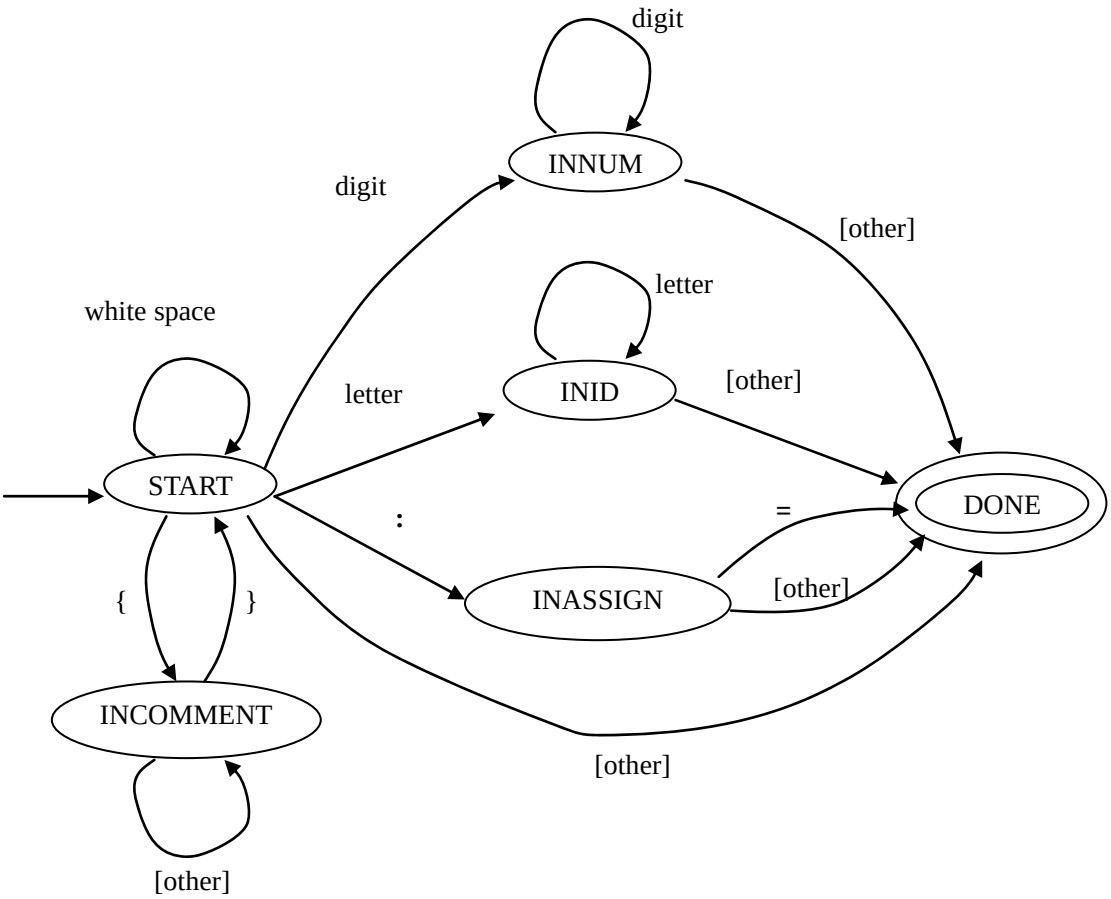
在该图中，不同的接受状态是由扫描程序返回的记号区分开来。如果在这个将要返回的记号（代码中的一个变量）中使用其他指示器，则所有接受状态都可集中为一个状态，称之为 DONE。若将这个二状态的 DFA 与接受数和标识符的 DFA 合并在一起，就可得到下面的

DFA：



请注意，利用方括号指出了不可被消耗的先行字符。

现在需要在这个 DFA 中添加注释、空白格和赋值。一个简单的从初始状态到其本身的循环要消耗空白格。注释要求一个额外的状态，它由花括号左边达到并在花括号右边返回到它。赋值也需要中间状态，它由分号上的初始状态达到。如果后面紧跟有一个等号，那么就会生成一个赋值记号。反之就不消耗下一个字符，且生成一个错误记号。实际上，未列在特殊符号中的所有单个字符既不是空白格或注释，也不是数字或字母，它们应被作为错误而接受，我们将它们与单个字符符号混合在一起。下图是为扫描程序给出的最后一个 DFA。



在上面的讨论或图中的 DFA 都未包括保留字。这是因为根据 DFA 的观点，而认为保留字与标识符相同，以后再在接受后的保留字表格中寻找标识符是最简单的。当然，最长子串原则保证了扫描程序唯一需要改变的动作是被返回的记号。因而，仅在识别了标识符之后才考虑保留字。

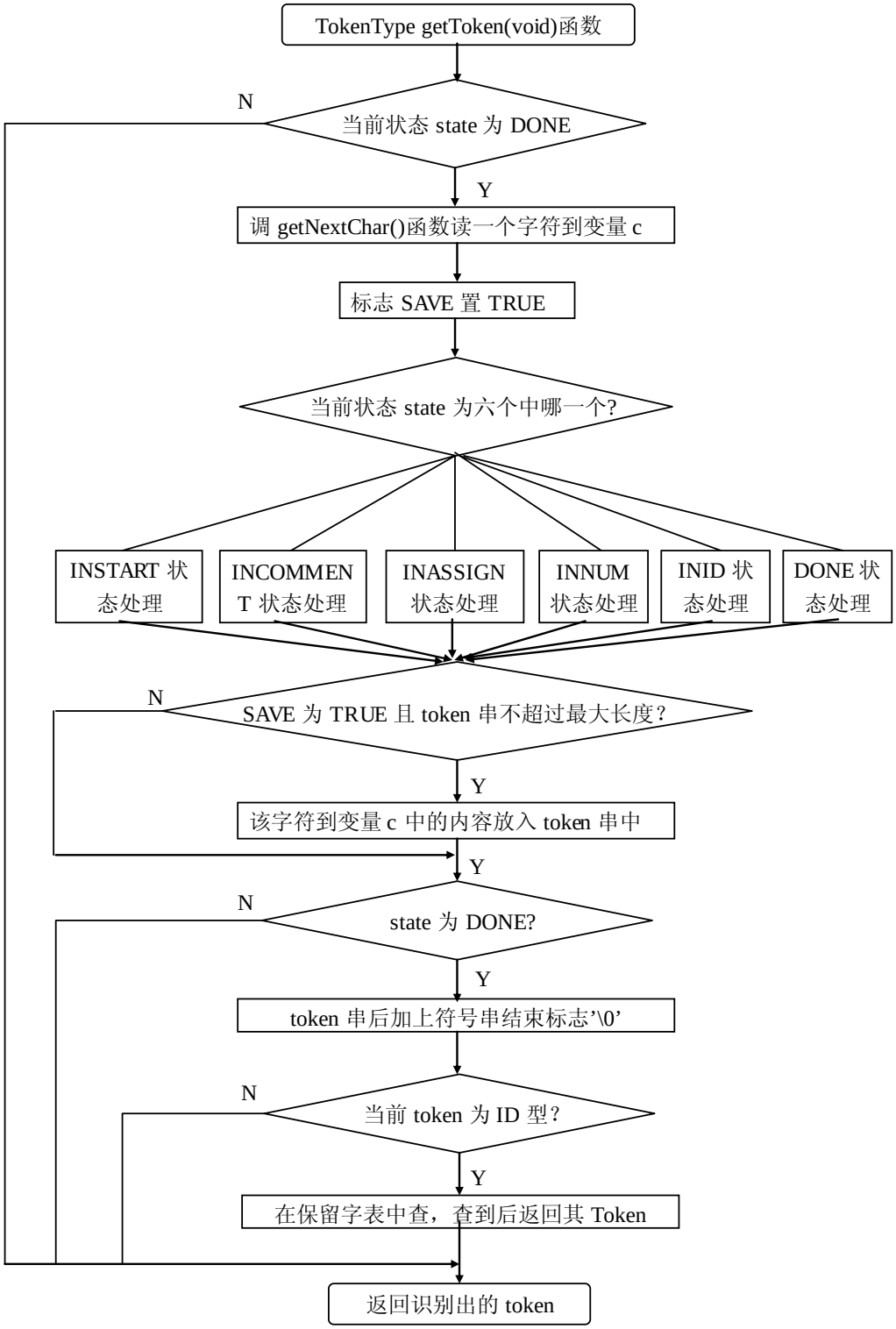
关于自动机 DFA 的相关定义：
在 scan.c 文件中定义：

```
/* states in scanner DFA */
typedef enum
    { START,INASSIGN,INCOMMENT,INNUM,INID,DONE }
    StateType;

/* lexeme of identifier or reserved word */
char tokenString[MAXTOKENLEN+1];

/* BUFLEN = length of the input buffer for
    source code lines */
#define BUFLEN 256

static char lineBuf[BUFLEN]; /* holds the current line */
static int linepos = 0; /* current position in LineBuf */
static int bufsize = 0; /* current size of buffer string */
static int EOF_flag = FALSE; /* corrects ungetNextChar behavior on EOF */
```

4、相关函数设计

现在再来讨论实现这个 DFA 的代码，它已被放在了 scan.h 文件和 scan.c 文件之中。

1、其中最主要的过程是 **getToken**（第 674 到第 793 行），它消耗输入字符并返回下一个被识别的记号。



2、这个实现利用了在 2.3.3 节中曾提到过的双重嵌套情况分析，以及一个有关状态的大型情况列表，在大列表中的是基于当前输入字符的单独列表。

3、记号本身被定义成 globals.h（第 174 行到第 186 行）中的枚举类型，它包括所有记号以及内务记号 EOF（当达到文件的末尾时）和 ERROR（当遇到错误字符时）。扫描程序的状态也被定义为一个枚举类型，但它是位于扫描程序之中（第 612 行到第 614 行）。

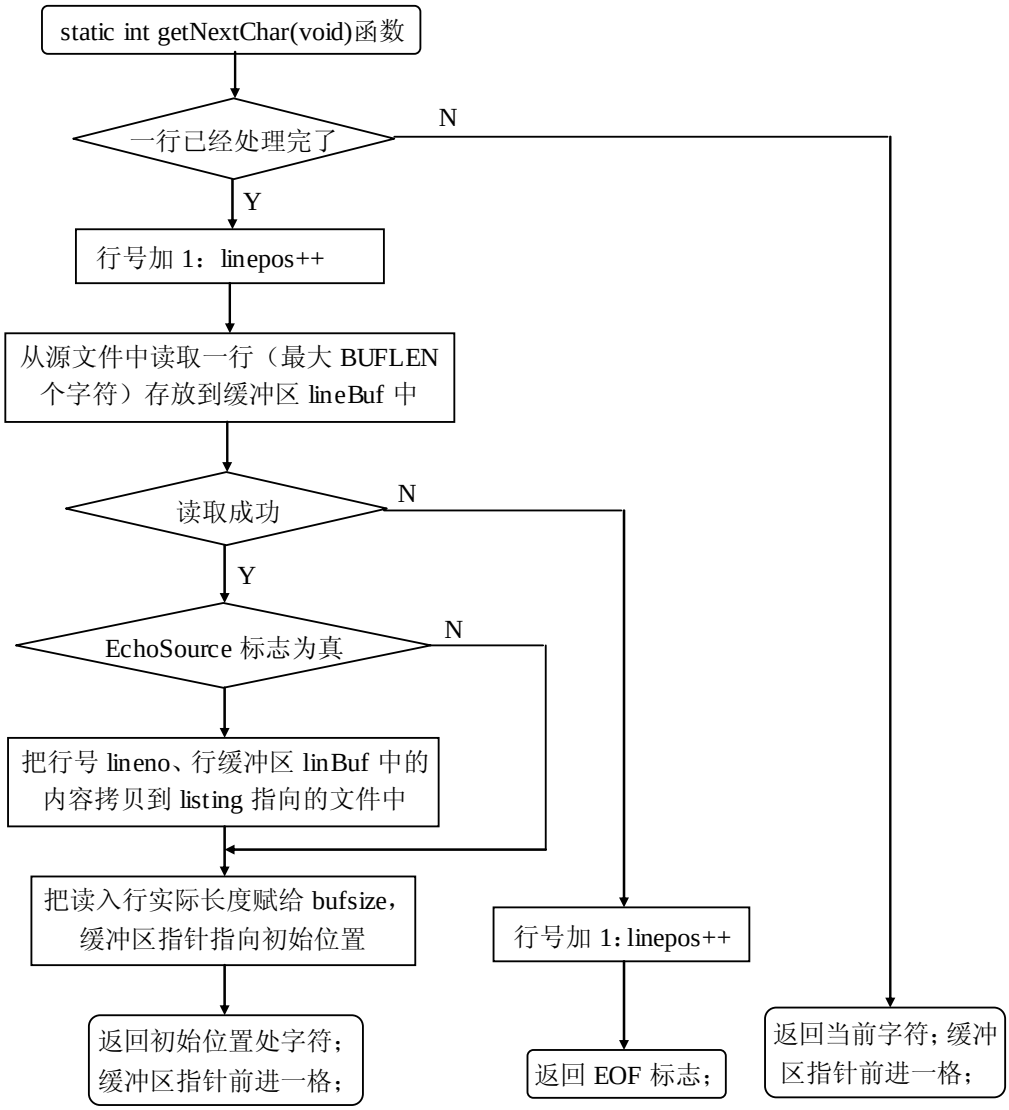
4、在 TINY 扫描程序中，所要计算的唯一特性是词法或是被识别的记号的串值，它位于变量 tokenString 之中。这个变量同 getToken 一并是提供给编译器其他部分的唯一的两个服务，

它们的定义已被收集在头文件 scan.h（第 550 行到第 571 行）。请读者注意声明了 tokenString 的长度固定为 41，因此那个标识符也就不能超过 40 个字符（加上结尾的空字符）。后面还会提到这个限制。

5、扫描程序使用了 3 个全程变量：文件变量 source 和 listing，在 globals.h 中声明且在 main.c 中被分配和初始化的整型变量 lineno。

6、由 getToken 过程完成的额外的簿记如下所述：表 reservedWords（第 649 行到第 656 行）和过程 reservedLookup（第 658 行到第 666 行）完成位于由 getToken 的主要循环识别的标识符之后的保留字的查找，currentToken 的值也随之改变。标志变量 save 被用作指示是否将一个字符增加到 tokenString 之上；由于需要包括空白格、注释和非消耗的先行，所以这些都是必要的。

7、到扫描程序的字符输入由 getNextChar 函数（第 627 行到第 642 行）提供，该函数将一个 256 -字符缓冲区内部的 lineBuf 中的字符取到扫描程序中。如果已经耗尽了这个缓冲区，且假设每一次都获取了一个新的源代码行（以及增加的 lineno），那么 getNextChar 就利用标准的 C 过程 fgets 从 source 文件更新该缓冲区。虽然这个假设允许了更简单的代码，但却不能正确地处理行的字数超过 255 个字符的 TINY 程序。在练习中，我们再探讨在这种情况下的 getNextChar 的行为（以及它更进一步的行为）。



8、可以通过提供一个 ungetNextChar 过程（第 644 行到第 647 行）在输入缓冲区中反填一个字符来完成这一任务

五、实验步骤

1、词法分析器设计

词法分析函数定义：
TokenType getToken(void)

显示分析结果的函数定义：
/* Procedure printToken prints a token
 * and its lexeme to the listing file
 */
void printToken(TokenType token, const char* tokenString)

2、命令行方式下手工编程词法分析器设计步骤

参见附件 1。

3、命令行方式下**Flex**自动工具词法分析器设计步骤

参见附件 2。

4、集成开发环境下词法分析器设计步骤

参见附件 3。

六、实验结果

输入：测试数据（以文件形式）；
输出：二元组（以文件形式）。

Figure 2.11 Sample program in the TINY language
{ sample program
In TINY language -
Computes factorial

```
   }
 read x; { input on integer }
 if    0 < x then    { don't compute if x <= 0 }
fact := 1 ;
repeat
fact := fact * x;
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

Figure 2.12 Output of scanner given the TINY program of figure 2.11 as input.

```
TINY COMPILATION: sample.tny
  1:    { Sample program
  2:      in TINY language –
  3:      computes factorial
  4:    }
  5:    read x; { input an integer }
5: reserved word: read
5: id, name= x
5: ;

  6: if 0 < x then { don't compute if x <= 0 }
6: reserved word: if
6: mum, val= 0
6: <
6: id, name= x
6: reserved word: then
7: fact := 1;
7: id, name= fact
7: :=
7: num, val= 1
7: ;
8: repeat
8: reserved word: repeat

9: fact := fact * x;
9: id, name= fact
9: :=
9: id, name= fact
9: *
9: id, name= x
9: ;
10:x := x - 1
```

10: id, name= x
10: :=
10: id, name=x
10: -
10: mum, val = 1
11:until x = 0;
11: reserved word: until
11: id, name= x
11: =
11: mum, val= 0
11: ;
12: write fact { output factorial of x }
12: reserved words: write
12: id, name= fact
13:end
13: reserved word: end
14: EOF

七、实验心得

1、遇到的难点与对策

2、体会

3、总结

第三章 Tiny语言语法分析器设计

一、实验目的

通过本次实验，学习 TINY 语言的上下文无关文法，进一步加深对递归下降算法与抽象语法树的理解，学习程序设计语言的语法分析器的手工编程方法。

二、实验内容

仔细阅读并测试 TINY 语言的语法分析器的相关程序，设计 TINY 语言语法分析器。

三、实验要求

1、TINY 语言的 BNF 文法，画出语法图

2、TINY 语言的抽象语法树节点的数据类型的定义与说明；

3、TINY 语言的语法分析器源程序的阅读与理解；

4、编译并测试 TINY 语言的语法分析器。

四、实验原理

1、TINY语言的上下文无关文法

下图是 TINY 在 BNF 中的文法，我们可从中观察到一些内容：TINY 程序只是一个语句序列，它共有 5 种语句：if 语句、repeat 语句、read 语句、write 语句和 assignment 语句。除了 if 语句使用 end 作为括号关键字（因此在 TINY 中没有悬挂 else 二义性）以及 if 语句和 repeat 语句允许语句序列作为主体之外，它们都具有类似于 Pascal 的语法，所以也就不需要括号或 begin-end 对（而且 begin 甚至在 TINY 中也不是一个保留字）。输入/输出语句由保留字 read 和 write 开始。read 语句一次只读出一个变量，而 write 语句一次只写出一个表达式。

program → *stmt-sequence*
stmt-sequence → *stmt-sequence* **;** *statement* | *statement*

statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt→ **if** *exp* **then** *stmt-sequence* **end**
| **if** *exp* **then** *stmt-sequence* **else** *stmt-sequence* **end**
repeat-stmt→ **repeat** *stmt-sequence* **until** *exp*
assign-stmt→ **identifier :=** *exp*
read-stmt → **read** *identifier*
write-stmt → **write** *exp*
exp → *simple-exp comparison-op simple-exp* | *simple-exp*
comparison-op → < | =
simple-exp → *simple-exp addop term* | *term*
addop → +|-
term → *term mulop factor factor* | *factor*
mulop → *|/
factor → **(***exp***)** |**number** |**identifier**

TINY 表达式有两类：在 if 语句和 repeat 语句的测试中使用比较算符=和<的布尔表达式，以及包括标准整型算符+、-、*和/（它代表整型除法，有时也写作 div）的算术表达式（由文法中的 simple-exp 指出）。算术运算是左结合并有通常的优先关系。相反地，比较运算却是非结合的：每个没有括号的表达式只允许一种比较运算。比较运算比其他算术运算的优先权都低。

TINY 中的标识符指的是简单整型变量，它没有诸如数组或记录构成的变量。TINY 中也没有变量声明：它只是通过出现在赋值语句左边来隐式地声明一个变量。另外，它只有一个（全局）作用域，且没有过程或函数（因此也就没有调用）。

还要注意 TINY 的最后一个方面。语句序列必须包括将语句分隔开来的分号，且不能将分号放在语句序列的最后一个语句之后。这是因为 TINY 没有空语句（不同于 Pascal 和 C）。另外，我们将 stmt-sequence 的 BNF 规则也写作一个左递归规则，但却并不真正在意语句序列的结合性，这是因为意图很简单，只需按顺序执行就行了。因此，只要将 stmt-sequence 编写成右递归即可。这个观点也出现在 TINY 程序的语法树结构中，其中的语法序列不是由树而是由列表表示的。现在就转到这个结构的讨论上来。

2、TINY编译器的语法树结构

TINY 有两种基本的结构类型：语句和表达式。语句共有 5 类（if 语句、repeat 语句、assign 语句、read 语句和 write 语句），表达式共有 3 类（算符表达式、常量表达式和标识符表达式）。因此，语法树节点首先按照它是语句还是表达式来分类，接着根据语句或表达式的种类进行再次分类。树节点最大可有 3 个孩子的结构（仅在带有 else 部分的 if 语句中才需要它们）。语句通过同属域而不是使用子域来排序。

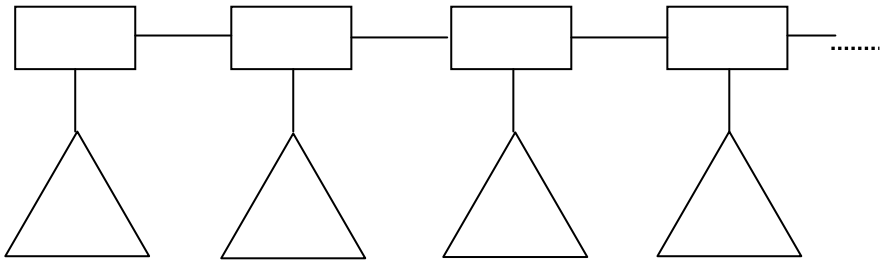
必须将树节点中的属性保留如下（除了前面所提到过的域之外）：每一种表达式节点都需要一个特殊的属性。常数节点需要它所代表的整型常数的域；标识符节点应包括了标识符名称的域；而算符节点则需要包括了算符名称的域。语句节点通常不需要属性（除了它们的节点类型之外）。但为了简便起见，在 assign 语句和 read 语句中，却要保留在语句节点本身中（除了作为一个表达式子节点之外）被赋予或被读取的变量名。

前面所描述的三个节点结构可通过程序清单 3 - 2 中的 C 说明得到，该说明还可在附录

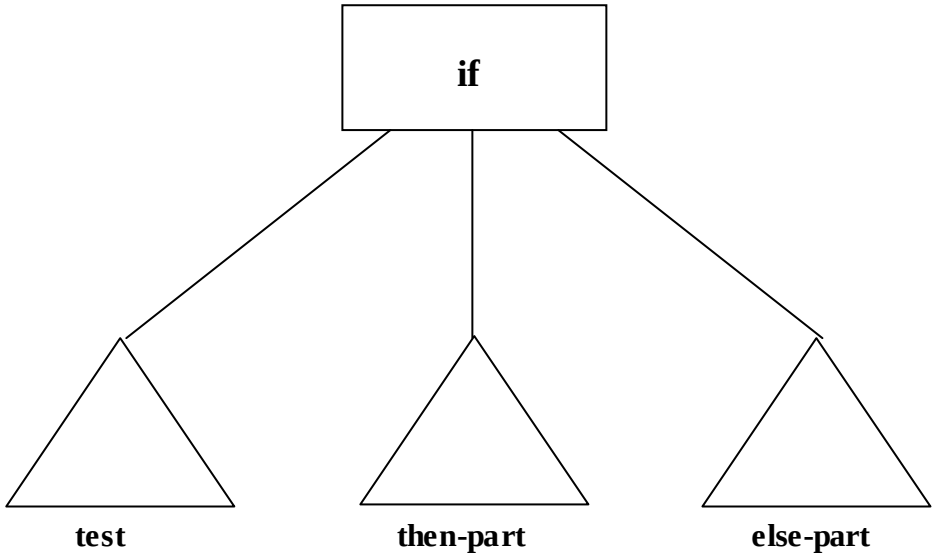
B（第 198 行到第 217 行）的 globals.h 文件的列表中找到。请注意我们综合了这些说明来帮助节省空间。它们还可帮助提醒每个节点类型的属性。现在谈谈说明中两个未曾提到过的属性。第 1 个是簿记属性 lineno；它允许在转换的以后步骤中出现错误时能够打印源代码行数。第 2 个是 type 域，在后面的表达式（且仅是表达式）类型检查中会用到它。

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,RepeatK,AssignK,ReadK,WriteK} StmtKind;
typedef enum {OpK,ConstK,IdK} ExpKind;

/* ExpType is used for type checking */
typedef enum {Void,Integer,Boolean} ExpType;

#define MAXCHILDREN 3

typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];
     struct treeNode * sibling;
     int lineno;
     NodeKind nodekind;
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
   } TreeNode;
```

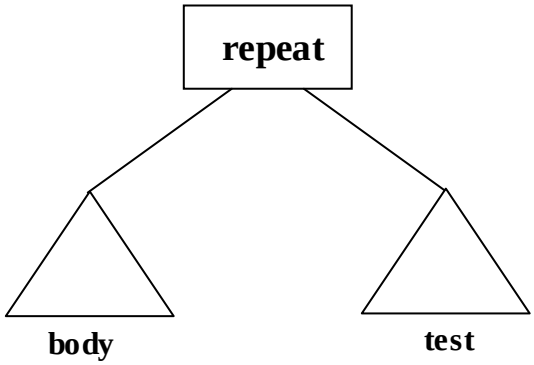
现在需要将语法树结构的描述用图形表示出来，并且画出示例程序的语法树。为了做到这一点，我们使用矩形框表示语句节点，用圆形框或椭圆形框表示表达式节点。语句或表达式的类型用框中的标记表示，额外的属性在括号中也列出来了。属性指针画在节点框的右边，而子指针则画在框的下面。我们还在图中用三角形表示额外的非指定的树结构，其中用点线表示可能出现也可能不出现的结构。语句序列由同属域连接（潜在的子树由点线和三角形表示）。则该图如下：



if 语句（带有 3 个可能的孩子）如下所示：

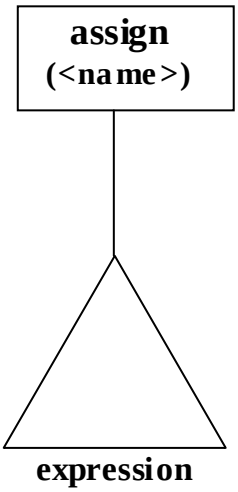


repeat 语句有两个孩子。第 1 个是表示循环体的语句序列，第 2 个是一个测试表达式：

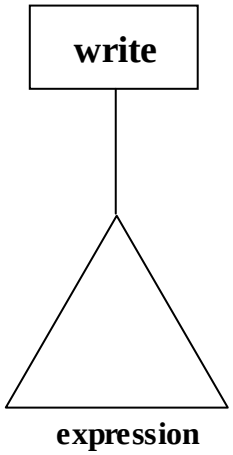


assign 语句有一个表示其值是被赋予的表达式的孩子(被赋予的变量名保存在语句节点

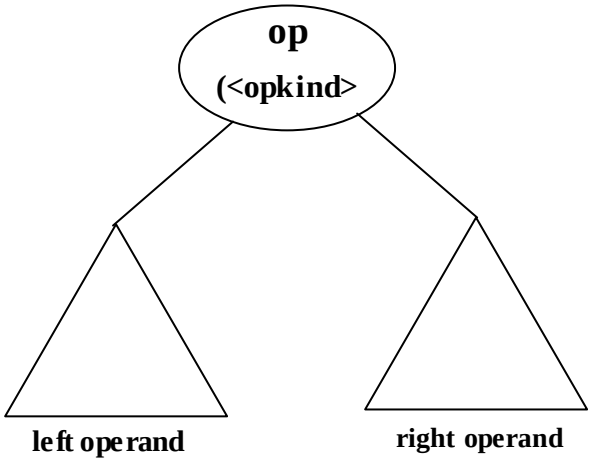
中）：



write 语句也有一个孩子，它表示要写出值的表达式：

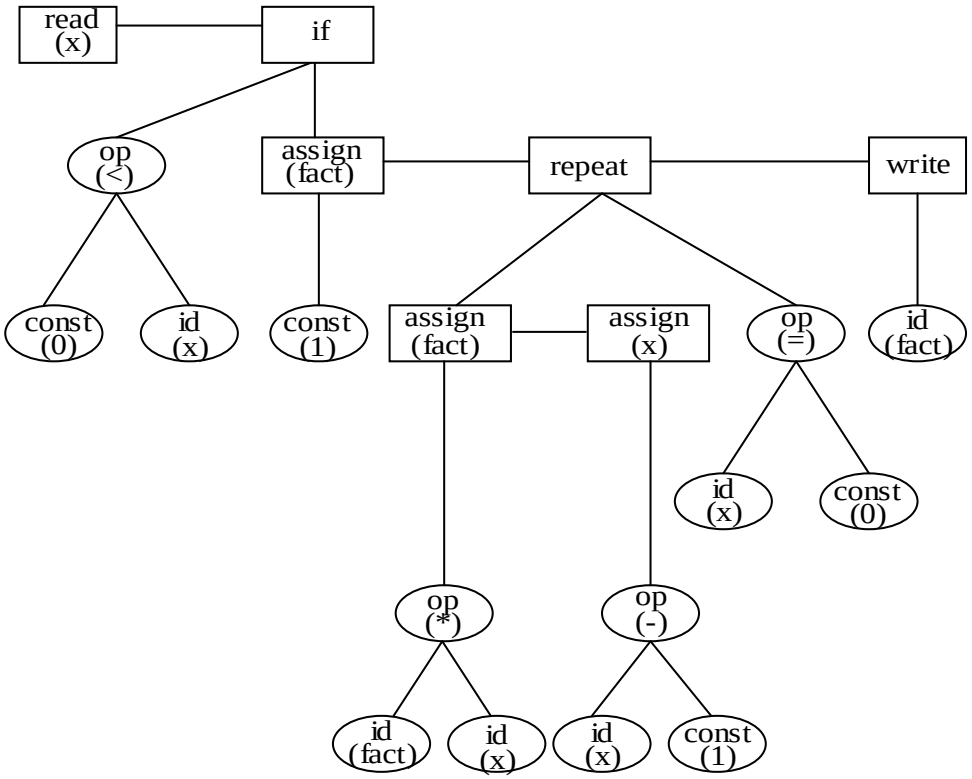


算符表达式有两个孩子，它们表示左操作数表达式和右操作数表达式：



其他所有的节点（read 语句、标识符表达式和常量表达式）都是叶子节点。

最后准备显示一个 TINY 程序的树。TINY 语言的示例程序为 sample.tny，它的语法树在下图中。



3、数据结构和全局变量

gloable.h 文件中定义：

（1）typedef enum
/* book-keeping tokens */
{ENDFILE, ERROR,
定义关键字：
/* reserved words */
IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE,
/* multicharacter tokens */
ID, NUM,
/* special symbols */
定义特殊符号：
ASSIGN, EQ, LT, PLUS, MINUS, TIMES, OVER, LPAREN, RPAREN, SEMI
} TokenType;

（2）几个相关的全局变量：

gloable.h 文件中定义：
extern FILE* source; /* source code text file */
extern FILE* listing; /* listing output text file */
extern int lineno; /* source line number for listing */

scan.h 文件中定义：
/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40
/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];

PARSE.H 文件中定义：
TreeNode * parse(void);
语法分析函数。

（3）用于观看分析结果的全局变量
用于观看分析结果的全局变量保存在 GLOBALS.H 文件中

/**/
/*********** Flags for tracing ************/
/**/

显示源代码：
/* EchoSource = TRUE causes the source program to
* be echoed to the listing file with line numbers

```
 * during parsing
 */
extern int EchoSource;
```

显示词法分析结果：

```
/* TraceScan = TRUE causes token information to be
 * printed to the listing file as each token is
 * recognized by the scanner
 */
extern int TraceScan;
```

显示语法分析结果：

```
/* TraceParse = TRUE causes the syntax tree to be
 * printed to the listing file in linearized form
 * (using indents for children)
 */
extern int TraceParse;
```

4、相关函数设计

TINY 语言的语法分析程序的代码保存在 parse.h 和 parse.c 两个文件中。

1. parse.h 文件非常简单：它由一个声明组成，声明语法树节点类型的语法分析函数，如下：

```
TreeNode * parse (void);
```

这个声明定义了主分析例程 parse，parse 又返回一个指向由分析程序构造的语法树的指针。

2.字符匹配函数由 match 过程完成，如果找到匹配，就调用 getToken，否则就声明出错，将出错信息打印到列表文件中的 syntaxError。

```
static void match(TokenType expected)
{ if (token == expected) token = getToken();
  else {/*出错时，没有消耗掉当前记号*/
    syntaxError("unexpected token -> ");
    printToken(token,tokenString);
    fprintf(listing,"      ");
  }
```

```
}
```

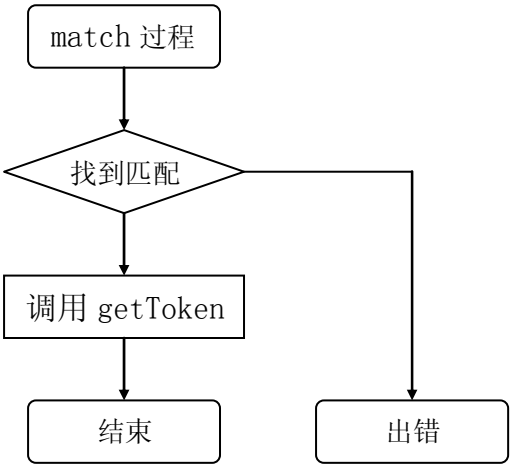


图 2-15 match 过程流程图

3. 语句函数为 TreeNode * statement(void)，它定义了 TINY 语言的 5 种不同语句的递归过程。如下：

```
TreeNode * statement(void)
{ TreeNode * t = NULL;
  switch (token) {
    case IF : t = if_stmt(); break;
    case REPEAT : t = repeat_stmt(); break;
    case ID : t = assign_stmt(); break;
    case READ : t = read_stmt(); break;
    case WRITE : t = write_stmt(); break;
    default : syntaxError("unexpected token -> ");
              printToken(token,tokenString);
              token = getToken();/*消耗掉当前记号*/
              break;
  } /* end case */
  return t;
}
```

五、实验步骤

1、语法分析器设计

语法分析函数定义：
TreeNode * parse(void)

显示分析结果的函数定义：

```
static void syntaxError(char * message)
{ fprintf(listing,"\n>>> ");
   fprintf(listing,"Syntax error at line %d: %s",lineno,message);
   Error = TRUE;
}

static void match(TokenType expected)
{ if (token == expected) token = getToken();
   else {
      syntaxError("unexpected token -> ");
      printToken(token,tokenString);
      fprintf(listing,"           ");
   }
}
```

2、命令行方式下手工编程语法分析器设计步骤

参见附件 1。

3、命令行方式下YACC自动工具语法分析器设计步骤

参见附件 2。

4、集成开发环境下语法分析器设计步骤

参见附件 3。

六、实验结果

1、输入数据：请以下代码保存为文件形式 sample.tny

```
   { sample program
In TINY language    -
Computes    factorial
   }
 read x; { input on integer }
 if    0 < x then    { don't compute if x <= 0 }
fact := 1 ;
repeat
fact := fact * x;
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

2、在控制台方式下输入 parserbyhand sample.tny，结果如下：

```
TINY COMPILATION: sample.tny

Syntax tree:
   Read: x
   If
      Op: <
         Const: 0
         Id: x
      Assign to: fact
         Const: 1
      Repeat
         Assign to: fact
            Op: *
               Id: fact
               Id: x
         Assign to: x
            Op: -
               Id: x
               Const: 1
```

```
    Op: =
      Id: x
      Const: 0
  Write
    Id: fact
```

七、实验心得

1、遇到的难点与对策

2、体会

3、总结

第四章Tiny语言语义分析器设计

一、实验目的

通过本次实验，进一步加深对语义分析的理解，学习编译器的语义分析器的编程方法。基于上一章构造的 TINY 语法分析程序，开发 TINY 语言的语义分析程序代码，编写语义分析器。

二、实验内容

1、设计符号表处理程序；

2、类型检查与推论程序的设计；

3、编译并测试 TINY 语言的语义分析器。

三、实验要求

1、认真学习 TINY 的符号表的设计方法；

2、编写类型检查与推论程序；

3、编译并测试 TINY 语言的语义分析器。

四、实验原理

1、TINY的符号表

在 TINY 语义分析程序符号表的设计中，首先确定什么信息需要在符号表中保存。一般情况这些信息包括数据类型和作用域信息。因为 TINY 没有作用域信息，并且所有的变量都是整型，TINY 符号表不需要保存这些信息。然而，在代码产生期间，变量需要分配存储器地址，并且因为在语法树中没有说明，因此符号表是存储这些地址的逻辑位置。现在，地址可以仅仅看成是整数索引，每次遇到一个新的变量时增加。为使符号表更加有趣和有用，还使用符号表产生一个交叉参考列表，显示被访问变量的行号。

作为符号表产生信息的例子，考虑下列 T I N Y 程序的例子(加上了行号)：

```
1: { Sample program
2: in TINY language --
3: computes factorial
4: }
5: read x; { input an integer }
6: if 0 < x then { d'otn compute if x <= 0 }
7: fact := 1;
8: repeat
9: fact := fact * x;
10: x := x - 1
11: until x = 0;
12: write fact { output factorial of x }
13: end
```

这个程序的符号表产生之后，语义分析程序将输出(TraceAnalyze = True)下列信息到列出的文件中：

```
Symbol table:
Variable Name      Location      Line      Numbers

-------------------------------------------------
```

X	**0**	**5**	**6**	**9**	**10**	**10**	**11**
fact	**1**	**7**	**9**	**9**	**12**		

2、符号表的代码

符号表的代码包含在 symtab.h 和 symtab.c 文件中。

符号表使用的结构是分离的链式杂凑表，因为没有作用域信息，所以不需要 delete 操作，insert 操作除了标识符之外，也只需要行号和地址参数。需要的其他的两个操作是打印刚才列出的文件中的汇总信息，以及 lookup 操作，从符号表中取出地址号(后面的代码产生器需要，符号表生成器也要检查是否已经看见了变量)。因此，头文件 symtab.h 包含下列说明：

```
void st_insert ( char * name, int lineno, int loc );
int st_lookup ( char * name );
void printSymTab(FILE * listing);
```

因为只有一个符号表，它的结构不需要在头文件中说明，也无须作为参数在这些过程中出现。

在 symtab.c 中相关的实现代码使用了一个动态分配链表，类型名是 LineList (第 1236 行到第 1239 行)，存储记录在杂凑表中每个标识符记录的相关行号。标识符记录本身保存在一个“桶”列表中，类型名是 BucketList (第 1247 行到第 1252 行)。stinsert 过程在每个“桶”列表(第 1262 行到第 1295 行)前面增加新的标识符记录，但行号在每个行号列表的尾部增加，以保持行号的顺序(stinsert 的效率可以通过使用环形列表或行号列表的前/后双向指针来改进)。

3、TINY语义分析程序

TINY 的静态语义共享标准编程语言的特性，符号表的继承属性，而表达式的数据类型是合成属性。因此，符号表可以通过对语法树的前序遍历建立，类型检查通过后序遍历完成。虽然这两个遍历能容易地组合成一个遍历，为使两个处理步骤操作的不同之处更加清楚，仍把它们分成语法树上两个独立的遍。因此，语义分析程序与编译器其他部分的接口，放在文件 analyze.h 中(附录 B，第 1350 行到第 1370 行)，由两个过程组成，通过下列说明给出

```
void buildSymtab(TreeNode *);
void typeCheck(TreeNode *);
```

第 1 个过程完成语法树的前序遍历，当它遇到树中的变量标识符时，调用符号表 stinsert 过程。遍历完成后，它调用 printSymTab 打印列表文件中存储的信息。第 2 个过程完成语法树的后序遍历，在计算数据类型时把它们插入到树节点，并把任意的类型检查错误记录到列表文件中。这些过程及其辅助过程的代码包含在 analyze.c 文件中（第 1400 行到第 1558 行）。

为强调标准的树遍历技术，实现 buildSymtab 和 typeCheck 使用了相同的通用遍历函数 traverse (第 1420 行到第 1441 行) ,它接受两个作为参数的过程(和语法树)，一个完成每个节

点的前序处理，一个进行后序处理：

```
static void traverse ( TreeNode * t,
void (* preProc) (TreeNode * ),
void (* postProc) (TreeNode * ) )
{ if (t != NULL)
{ preProc(t);
{ int i;
for (i=0; i < MAXCHILDREN; i++)
traverse(t->child[i],preProc, postProc);
}
postProc (t) ;
traverse(t->sibling, preProc, postProc);
}
}
```

给定这个过程，为得到一次前序遍历，当传递一个“什么都不做”的过程作为 preproc 时，需要说明一个过程提供前序处理并把它作为 preproc 传递到 traverse。对于 TINY 符号表的情况，前序处理器称作 insertNode，因为它完成插入到符号表的操作。“什么都不做”的过程称作 nullProc，它用一个空的过程体说明(第 1438 行到第 1441 行)。然后建立符号表的前序遍历由 buildSymtab 过程(第 1488 行到第 1494 行)内的单个调用

```
traverse (syntaxTree, insertNode, nullProc);
```

完成。类似地，typeCheck (第 1556 行到第 1558 行)要求的后序遍历由单个调用

```
traverse (syntaxTree, nullProc, checkNode);
```

完成。这里 checkNode 是一个适当说明的过程，计算和检查每个节点的类型。现在还剩下描述过程 insertNode 和 checkNode 的操作。

insertNode 过程(第 1447 行到第 1483 行)必须基于它通过参数(指向语法树节点的指针)接受的语法树节点的种类，确定何时把一个标识符(与行号和地址一起)插入到符号表中。对于语句节点的情况，包含变量引用的节点是赋值节点和读节点，被赋值或读出的变量名包含在节点的 attr.name 字段中。对表达式节点的情况，感兴趣的是标识符节点，名字也存储在 attr.name 中。因此，在那 3 个位置，如果还没有看见变量 insertNode 过程包含一个

```
st_insert (t->attr.name, t->lineno, location++);
```

调用(与行号一起存储和增加地址计数器)，并且如果变量已经在符号表中，则

```
st_insert (t->attr.name,t->lineno,0);
```

(存储行号但没有地址)。

最后，在符号表建立之后，buildSymtab 完成对 printSymTab 的调用，在标志 TracaAnalyze 的控制下(在 m a i n . c 中设置)，在列表文件中写入行号信息。

类型检查遍的 checkNode 过程有两个任务。首先，基于子节点的类型，它必须确定是否出现了类型错误。其次，它必须为当前节点推断一个类型(如果它有一个类型)并且在树节点中为这个类型分配一个新的字段。这个字段在 TreeNode 中称作 type 字段(在 globals.h 中说明，第 216 行)。因为仅有表达式节点有类型，这个类型推断只出现在表达式节点。在 TINY 中只有两种类型，整型和布尔型，这些类型在全局说明的枚举类型中说明(第 203 行)：

typedef enum {Void, Integer, Boolean} ExpType;

这里类型 Void 是“无类型”类型，仅用于初始化和错误检查。当出现一个错误时，

checkNode 过程调用 typeError 过程，基于当前的节点，在列表文件中打印一条错误消息。

还剩下归类 checkNode 的动作。对表达式节点，节点可以是叶子节点(常量或标识符，种类是 ConstK 或 IdK)，或者是操作符节点(种类 OpK)。对叶子节点的情况(第 1517 行到第 1520 行)，类型总是 Integer (没有类型检查发生)。对操作符节点的情况(第 1508 行到第 1516 行)，两个子孙子表达式的类型必须是 Integer (因为后序遍历已经完成，已经计算出它们的类型)。然后，OpK 节点的类型从操作符本身确定(不关心是否出现了类型错误)：如果操作符是一个比较操作符(<或=)，那么类型是 Boolean；否则是 Integer。

对语句节点的情况，没有类型推断，但除了一种情况，必须完成某些类型检查。这种情况是 ReadK 语句，这里被读出的变量必须自动成为 Integer 类型，因此没有必要进行类型检查。所有 4 种其他语句种类需要一些形式的类型检查： If K 和 RepeatK 语句需要检查它们的测试表达式，确保它们是类型 Boolean(第 1527 行到第 1530 行和第 1539 行到第 1542 行)，而 WriteK 和 AssignK 语句需要检查(第 1531 行到第 1538 行)确定被写入或赋值的表达式不是布尔型的(因为变量只能是整型值，只有整型值能被写入)：

```
x := 1 < 2; { error - Boolean value
cannot be assigned }
write 1 = 2; { also an error }
```

4、数据结构和全局变量

gloable.h 文件中定义：

（1）typedef enum

```
/* book-keeping tokens */
{ENDFILE, ERROR,
 定义关键字：
 /* reserved words */
 IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE,
 /* multicharacter tokens */
 ID, NUM,
 /* special symbols */
 定义特殊符号：
 ASSIGN, EQ, LT, PLUS, MINUS, TIMES, OVER, LPAREN, RPAREN, SEMI
} TokenType;
```

（2）几个相关的全局变量：

gloable.h 文件中定义：

```
extern FILE* source; /* source code text file */
extern FILE* listing; /* listing output text file */
```

```
extern FILE* code; /* code text file for TM simulator */
extern int lineno; /* source line number for listing */
```

scan.h 文件中定义：

```
/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40
/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];
```

ANALYZE.H 文件中定义：

```
/* Function buildSymtab constructs the symbol
 * table by preorder traversal of the syntax tree
 */
void buildSymtab(TreeNode *);

/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode *);
```

（3）用于观看分析结果的全局变量

用于观看分析结果的全局变量保存在 GLOBALS.H 文件中

```
/*****************************************************/
/***********      Flags for tracing            ************/
/*****************************************************/

/* EchoSource = TRUE causes the source program to
 * be echoed to the listing file with line numbers
 * during parsing
 */
extern int EchoSource;

/* TraceScan = TRUE causes token information to be
 * printed to the listing file as each token is
 * recognized by the scanner
 */
extern int TraceScan;

/* TraceParse = TRUE causes the syntax tree to be
 * printed to the listing file in linearized form
 * (using indents for children)
 */
```

```
extern int TraceParse;

/* TraceAnalyze = TRUE causes symbol table inserts
 * and lookups to be reported to the listing file
 */
extern int TraceAnalyze;

/* TraceCode = TRUE causes comments to be written
 * to the TM code file as code is generated
 */
extern int TraceCode;
```

5、相关函数设计

TINY 语言的语义分析程序的代码保存在 analyze.h 和 analyze.c 两个文件中。

analyze.h 由两个函数组成，如下：

```
/* Function buildSymtab constructs the symbol
 * table by preorder traversal of the syntax tree
 */
void buildSymtab(TreeNode *);

/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode *);
```

五、实验步骤

1、语义分析器设计

语义分析函数定义：

```
/* Function buildSymtab constructs the symbol
 * table by preorder traversal of the syntax tree
 */
void buildSymtab(TreeNode *);

/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode *);
```

显示分析结果的函数定义：

```
void buildSymtab(TreeNode * syntaxTree)
{ traverse(syntaxTree,insertNode,nullProc);
  if (TraceAnalyze)
  { fprintf(listing,"\nSymbol table:\n\n");
    printSymTab(listing);
  }
}
```

显示出错信息：

```
static void typeError(TreeNode * t, char * message)
{ fprintf(listing,"Type error at line %d: %s\n",t->lineno,message);
  Error = TRUE;
}
```

2、命令行方式下手工编程语义分析器设计步骤

参见附件 1。

3、集成开发环境下语义分析器设计步骤

参见附件 2。

六、实验结果

1、输入数据：请以下代码保存为文件形式 sample.tny

```
{ sample program
In TINY language    -
Computes    factorial
   }
 read x; { input on integer }
 if    0 < x then    { don't compute if x <= 0 }
fact := 1 ;
repeat
fact := fact * x;
x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

2、在控制台方式下输入 AnalyzerByHand sample.tny，结果如下：

```
TINY COMPILATION: sample.tny

Building Symbol Table...

Symbol table:

Variable Name    Location      Line Numbers
-------------    --------      ------------
x                    0                5       6       9     10      10      11
fact                 1                7       9       9     12

Checking Types...

Type Checking Finished
```

七、实验心得

1、遇到的难点与对策

2、体会

3、总结

第六章 Tiny语言代码生成器设计

本节首先描述 TINY 代码生成器和 TM 的接口以及代码生成必需的实用函数。然后再说明代码生成的步骤。接着，描述 TINY 编译器和 TM 模拟器的结合。最后讨论全书使用的示例 TINY 程序的目标代码。

一、实验目的

（1）了解 TINY 代码生成器的 TM 接口；
（2）了解代码生成器的内部工作原理；
（3）进一步巩固命令行方式和图形用户界面方式下的程序设计方法；

二、实验内容

代码生成是 Tiny 编译器的最后一项工作，代码生成的基础是语法树和符号表，遍历语法树，生成能够被 TM 虚拟机执行的指令，其中 if 语句和 repeat 语句需要利用 emitSkip、emitBackup 进行代码回填，因为只有全部语句指令生成完成以后才知道跳转地址。

用 TINY 编译器产生和使用 TM 代码文件。

三、 实验要求

要求实现编译器的以下功能：生成可以被 TM 模拟器识别的目标代码。

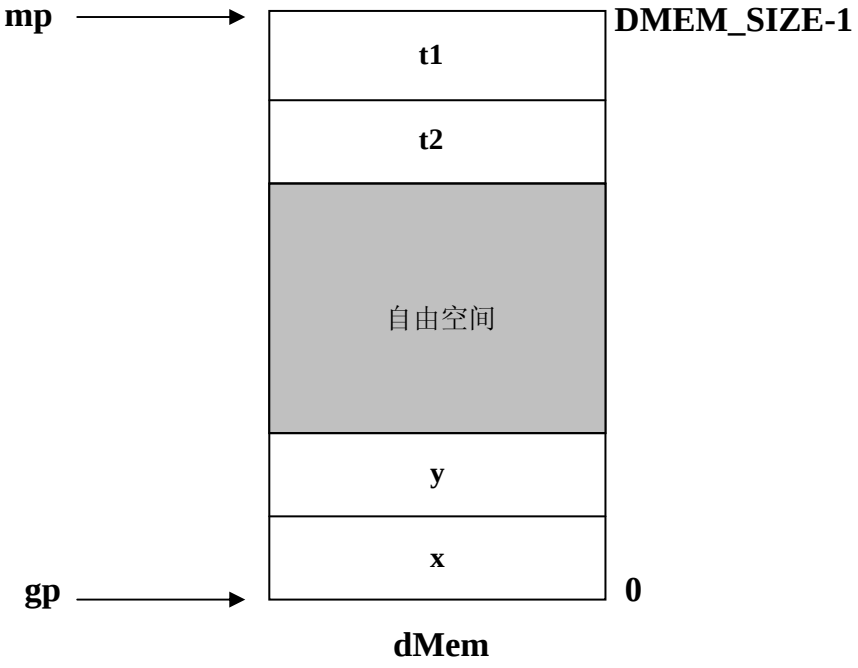
四、实验原理

1、TINY代码生成器的TM接口

一些代码生成器需要知道的有关 TM 的信息已封装在文件 code.h 和 code.c 中，分别是第 1600 行到第 1685 行和第 1700 行到第 1796 行。此外还在文件中放入了代码发行函数。当然，代码生成器还是要知道 TM 指令的名字，但是这些文件分离了指令格式的详细说明和目标代码文件的位置以及运行时使用特殊寄存器。code.c 文件完全可以将指令序列放到特别的

iMem 位置，而代码生成器就不必追踪细节了。如果 TM 装载器要改进，也就是说允许符号标号并去掉数字编号，那么将很容易将标号生成和格式变化加入到 code.c 文件中。

首先是寄存器值的定义(1612，1617，1623，1626 和 1629 行)。明显地，代码生成器和代码发行实用程序必须知道 pc。另外还有 TINY 语言的运行时环境，如前所述，将数据存储时的顶部分配给临时存储(以栈方式)而底部则分配给变量。由于 TINY 中没有活动记录(于是也就没有 fp) (没有作用域和过程调用)，变量和临时存储的位置可认为是绝对的。然而，TM 机的 LD 操作不允许绝对地址，而必须有一个寄存器基值来计算存储装入的地址。这样我们分配两个寄存器，称为 mp (内存指针)和 gp (全程指针)来指示存储区的顶部和底部。mp 将用于访问临时变量，并总是包含最高正规内存位置，而 gp 用于所有命名变量访问，并总是包含 0。这样由符号表计算的绝对地址可以生成相对 gp 的偏移来使用。例如，如果程序使用两个变量 x 和 y，并有两个临时值存在内存中，那么 dMem 将如下所示：



在本图中，t1 的地址为 0 (mp)，t2 为- 1 (mp)，x 的地址为 0 (gp)，而 y 为 1 (gp)。在这个实现中，gp 是寄存器 5，mp 是寄存器 6。

另两个代码生成器将使用的寄存器是寄存器 0 和 1，称之为“累加器”并命令名为 ac 和 ac1。它们被当作相等的寄存器来使用。通常计算结果存放在 ac 中。注意寄存器 2、3 和 4 没有命名(且从不使用 1)。

现在来讨论 7 个代码发行函数，原型在 code.h 文件中给出。如果 TraceCode 标志置位，emitComment 函数会以注释格式将其参数串打印到代码文件中的新行中。下两个函数 emitRO 和 emitRM 为标准的代码发行函数用于 RO 和 RM 指令类。除了指令串和 3 个操作数之外，每个函数还带有 1 个附加串参数，它被加到指令中作为注释(如果 TraceCode 标志置位)。

接下来的 3 个函数用于产生和反填转移。emitSkip 函数用于跳过将来要反填的一些位置

并返回当前指令位置且保存在 code.c 内部。典型的应用是调用 emitSkip (1)，它跳过一个位置，这个位置后来会填上转移指令，而 emitSkip (0)不跳过位置，调用它只是为了得到当前位置以备后来的转移引用。函数 emitBackup 用于设置当前指令位置到先前位置来反填，emitRestore 用于返回当前指令位置给先前调用 emitBackup 的值。典型地，这些指令在一起使用如下：

```
emitBackup(savedLoc) ;
/* generate backpatched jump instruction here */
emitRestore() ;
```

最后代码发行函数(emitRMAbs)用来产生诸如反填转移或任何由调用 emitSkip 返回的代码位置的转移的代码。它将绝对代码地址转变成 pc 相关地址，这由当前指令位置加 1 (这是 pc 继续执行的地方)减去传进的位置参数，并且使用 pc 做源寄存器。通常地，这个函数仅用于条件转移，比如 JEQ 或使用 LDA 和 pc 作为目标寄存器产生无条件转移， 如前一小节所述的那样。

这样就描述完了 TINY 代码生成实用程序，我们来看一看 tiny 代码生成器本身的描述。

2、TINY代码生成器

TINY 代码生成器在文件 cgen.c 中，其中提供给 TINY 编译器的唯一接口是 CodeGen，其原型为：

```
void CodeGen (void);
```

在接口文件 cgen.h 中给出了唯一的定义。参见第 1900 行到第 2111 行。

函数 CodeGen 本身(第 2095 行到第 2111 行)所做的事极少：产生一些注释和指令(称为标准序言(standard prelude))、设置启动时的运行时环境，然后在语法树上调用 cG e n，最后产生 HALT 指令终止程序。标准序言由两条指令组成：第 1 条将最高正规内存位置装入 mp 寄存器(TM 模拟器在开始时置 0)。第 2 条指令清除位置 0 (由于开始时所有寄存器都为 0，gp 不必置 0)。

函数 cGen(第 2070 行到第 2084 行)负责完成遍历并以修改过的顺序产生代码的语法树，回想 TINY 语法树定义给出的格式：

```
typedef enum { StmtK, ExpK } NodeKind;
typedef enum { IfK, RepeatK, AssignK, ReadK, WriteK } StmtKind;
typedef enum { OpK, ConstK, IdK } ExpKind;
#define MAXCHILDREN 3
typedef struct treeNode
{ struct treeNode * child[MAXCHILDREN] ;
struct treeNode * sibling;
int lineno;
NodeKind nodekind;
union { StmtKind stmt; ExpKind exp; } kind;
union { TokenType op;
int val;
```

```
char * name; } attr;
ExpType type;
} TreeNode;
```

这里有两种树节点：句子节点和表达式节点。如果节点为句子节点，那么它代表 5 种不同 TINY 语句(if、repeat、赋值、read 或 write)中的一种，如果节点为表达式节点，则代表 3 种表达式(标识符、整形常数或操作符)中的一种。函数 cGen 仅检测节点是句子或表达式节点(或空)，调用相应的函数 genStmt 或 genExp，然后在同属上递归调用自身(这样同属列表将以从左到右格式产生代码)。

函数 genStmt (第 1924 行到第 1994 行)包含大量 switch 语句来区分 5 种句子，它产生代码并在每种情况递归调用 cGen，genExp 函数(第 1997 行到第 2065 行)也与之类似。在任意情况下，子表达式的代码都假设把值存到 ac 中而可以被后面的代码访问。当需要访问变量时(赋值和 read 语句以及标识符表达式)，通过下面操作访问符号表：

```
loc = lookup(tree->attr.name);
```

loc 的值为问题中的变量地址并以 gp 寄存器基准的偏移装入或存储值。

其他需要访问内存的情况是计算操作符表达式的结果，左边的操作数必须存入临时变量直到右边操作数计算完成。这样操作符表达式的代码包含下列代码生成序列在操作符应用(第 2021 行到第 2027 行)之前：

```
cGen(p1); /* p1 = left child */
emitRM("ST",ac,tmpOffset--,mp,"op: push left");
cGen(p2); /* p2 = right child */
emitRM("LD",ac1,++tmpOffset,mp,"op: load left");
```

这里的 tmpOffset 为静态变量，初始为用作下一个可用临时变量位置对于内存顶部(由 m p 寄存器指出)的偏移。注意 tmpOffset 如何在每次存入后递减和读出后递增。这样 tmpOffset 可以看成是“临时变量栈”的顶部指针，对 emifRM 函数的调用与压入和弹出该栈相对应。这在临时变量在内存中时保护它们。在以上代码之前执行实际动作，左边的操作数将在寄存器 1 (acl)中而右边操作数在寄存器 0 (ac)中。如果是算术操作的话，就产生相应的 RO 操作。

比较操作符的情况有少许差别。TINY 语言的语法(如语法分析器中的实现，参见前面章节)仅在 if 语句和 while 语句的测试表达式中允许比较操作符。在这些测试之外也没有布尔变量或值，比较操作符可以在这些语句的代码生成内部处理。然而，这里我们用更通常的方法，它更广泛应用于包含逻辑操作与/或布尔值的语言，并将测试结果表示为 0 (假)或 1 (真)，如同在 C 中一样。这要求常数据 0 或 1 显式地装入 ac，用转移到执行正确装载来实现这一点。例如，在小于操作符的情况中，产生了以下代码，代码产生将计算左边操作数存入寄存器 1，并计算右边操作数存入寄存器 0：

```
SUB 0,1,0
JLT 0,2(7)
LDC 0,0(0)
LDA 7,1(7)
LDC 0,1(0)
```

第 1 条指令将左边操作数减去右边操作数，结果放入寄存器 0，如果< 为真，结果应为负值，并且指令 JLT 0,2(7)将导致跳过两条指令到最后一条，将值 1 装入 ac，如果< 为假，将执行第 3 条和第 4 条指令，将 0 装入 a c 然后跳过最后一条指令(回忆 TM 的描述，LDA 使用 pc 为寄存器引起无条件转移)。

我们将以 if -语句(第 1930 行到第 1954 行)的讨论来结束 TINY 代码生成器的描述。

代码生成器为 if 语句所做的第 1 个动作是为测试表达式产生代码。如前所述测试代码，在假时将 0 存入 ac，真时将 1 存入。生成代码接下来要产生一条 JEQ 到 if 语句的 else 部分。然而这些代码的位置当前是未知的，这是因为 t h e n 部分的代码还要生成。因此，代码生成器用 emitSkip 来跳过后面语句并保存位置用于反填：

```
savedLoc1 = emitSkip(1) ;
```

代码生成继续处理 if 算语句的 then 部分。之后必须无条件转移跳过 else 部分。同样转移位置未知，于是这个转移的位置也要跳过并保存位置：

```
savedLoc2 = emitSkip(1) ;
```

现在，下一步是产生 else 部分的代码，于是当前代码位置是正确的假转移的目标，要反填到位置 savedLoc1。下面的代码处理之：

```
currentLoc = emitSkip (0) ;
emitBack up(savedLoc1) ;
emitRM_Abs("JEQ",ac,currentLoc,"if: jmp to else") ;
emitRestors() ;
```

注意 emitSkip 调用是如何用来获取当前指令位置的，以及 emitRMAbs 过程如何用于将绝对地址转移变换成 p c 相关的转移，这是 JEQ 指令所需的。之后就可以为 else 部分产生代码了，然后用类似的代码将绝对转移(LDA)反填到 savedLoc2。

3、用TINY编译器产生和使用TM代码文件

TINY 代码生器可以合谐地与 TM 模拟器一起工作。当主程序标志 NO_ PARSE 、NOANALYZE 和 NOCODE 都置为假时，编译器创建. t m 后缀的代码文件(假设源代码中无错误)并将 T M 指令以 T M 模拟器要求的格式写入该文件。例如，为编译并执行 sample.t n y 程序，只要发出下面命令：

```
tiny sample
<listing produced on the standard output>
tm sample
<execution of the tm simulator>
```

为了跟踪的目的，有一个 TraceCode 标志在 globals.h 中声明，其定义在 main.c 中。如果标志为 TRUE，代码生成器将产生跟踪代码，在代码文件中表现为注释，指出每条指令或指令序列在代码生成器的何处产生以及产生原因。

4、TINY编译器生成的TM代码文件示例

为了详细说明代码生成是如何工作的，我们在下面展示了 TINY 代码生成器生成的示例程序的代码，由于 TraceCode=TRUE，所以也产生了代码注释。这个代码文件有 42 条指令，其中包括来自标准序言的两条指令。

```
* TINY Compilation to TM Code
* File: sample.tm
* Standard prelude:
0: LD 6,0(0) load maxaddress from location 0
1: ST 0,0(0) clear location 0
* End of standard prelude.
```

```
2: IN 0,0,0 read integer value
3: ST 0,0(5) read: store value
* -> if
* -> Op
* -> Const
4: LDC 0,0(0) load const
* <- Const
5: ST 0,0(6) op: push left
* -> Id
6: LD 0,0(5) load id value
* <- Id
7: LD 1,0(6) op: load left
8: SUB 0,1,0 op <
9: JLT 0,2(7) br if true
10: LDC 0,0(0) false case
11: LDA 7,1(7) unconditional jmp
12: LDC 0,1(0) true case
* <- Op
* if: jump to else belongs here
* -> assign
* -> Const
14: LDC 0,1(0) load const
* <- Const
15: ST 0,1(5) assign: store value
* <- assign
* -> repeat
* repeat: jump after body comes back here
* -> assign
* -> Op
* -> Id
16: LD 0,1(5) load id value
* <- Id
17: ST 0,0(6) op: push left
* -> Id
18: LD 0,0(5) load id value
* <- Id
19: LD 1,0(6) op: load left
20: MUL 0,1,0 op *
* <- Op
21: ST 0,1(5) assign: store value
* <- assign
* -> assign
* -> Op
* -> Id
```

```
22: LD 0,0(5) load id value
* <- Id
23: ST 0,0(6) op: push left
* -> Const
24: LDC 0,1(0) load const
* <- Const
25: LD 1,0(6) op: load left
26: SUB 0,1,0 op –
* <- Op
27: ST 0,0(5) assign: store value
* <- assign
* -> Op
* -> Id
28: LD 0,0(5) load id value
* <- Id
29: ST 0,0(6) op: push left
* -> Const
30: LDC 0,0(0) load const
* <- Const
31: LD 1,0(6) op: load left
32: SUB 0,1,0 op = =
33: JEQ 0,2(7) br if true
34: LDC 0,0(0) false case
35: LDA 7,1(7) unconditional jmp
36: LDC 0,1(0) true case
* <- Op
37: JEQ 0,-22(7) repeat: jmp back to body
* <- repeat
* -> Id
38: LD 0,1(5) load id value
* <- Id
39: OUT 0,0,0 write ac
* if: jump to end belongs here
13: JEQ 0,27(7) if: jmp to else
40: LDA 7,0(7) jmp to end
* <- if
* End of execution.
41: HALT 0,0,0
```

5、数据结构和全局变量

gloable.h 文件中定义：

（1）typedef enum
/* book-keeping tokens */
{ENDFILE, ERROR,
定义关键字：
/* reserved words */
IF, THEN, ELSE, END, REPEAT, UNTIL, READ, WRITE,
/* multicharacter tokens */
ID, NUM,
/* special symbols */
定义特殊符号：
ASSIGN, EQ, LT, PLUS, MINUS, TIMES, OVER, LPAREN, RPAREN, SEMI
} TokenType;

（2）几个相关的全局变量：

gloable.h 文件中定义：

```
extern FILE* source; /* source code text file */
extern FILE* listing; /* listing output text file */
extern FILE* code; /* code text file for TM simulator */
extern int lineno; /* source line number for listing */
```

scan.h 文件中定义：

```
/* MAXTOKENLEN is the maximum size of a token */
#define MAXTOKENLEN 40
/* tokenString array stores the lexeme of each token */
extern char tokenString[MAXTOKENLEN+1];
```

ANALYZE.H 文件中定义：

```
/* Function buildSymtab constructs the symbol
 * table by preorder traversal of the syntax tree
 */
void buildSymtab(TreeNode *);

/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode *);
```

（3）用于观看分析结果的全局变量
用于观看分析结果的全局变量保存在 GLOBALS.H 文件中

```
/*****************************************************/
/***********      Flags for tracing            ************/
/*****************************************************/

/* EchoSource = TRUE causes the source program to
 * be echoed to the listing file with line numbers
 * during parsing
 */
extern int EchoSource;

/* TraceScan = TRUE causes token information to be
 * printed to the listing file as each token is
 * recognized by the scanner
 */
extern int TraceScan;

/* TraceParse = TRUE causes the syntax tree to be
 * printed to the listing file in linearized form
 * (using indents for children)
 */
extern int TraceParse;

/* TraceAnalyze = TRUE causes symbol table inserts
 * and lookups to be reported to the listing file
 */
extern int TraceAnalyze;

/* TraceCode = TRUE causes comments to be written
 * to the TM code file as code is generated
 */
extern int TraceCode;
```

6、相关函数设计

TINY 语言的代码生成程序的代码保存在 cgen.h 和 cgen.c 两个文件中。

代码生成函数：

```
void codeGen(TreeNode * syntaxTree, char * codefile)
{   char * s = malloc(strlen(codefile)+7);
    strcpy(s,"File: ");
    strcat(s,codefile);
    emitComment("TINY Compilation to TM Code");
    emitComment(s);
    /* generate standard prelude */
    emitComment("Standard prelude:");
    emitRM("LD",mp,0,ac,"load maxaddress from location 0");
    emitRM("ST",ac,0,ac,"clear location 0");
    emitComment("End of standard prelude.");
    /* generate code for TINY program */
    cGen(syntaxTree);
    /* finish */
    emitComment("End of execution.");
    emitRO("HALT",0,0,0,"");
}
```

五、实验步骤

1、命令行方式下手工编程代码生成器设计步骤

参见附件 1。

2、集成开发环境下代码生成器设计步骤

参见附件 2。

六、实验结果

命令行方式下输入 cgen sample.tny 命令
将在 DEBUG 目录下生成 sample.tm 代码文件

七、实验心得

1、遇到的难点与对策
2、体会
3、总结