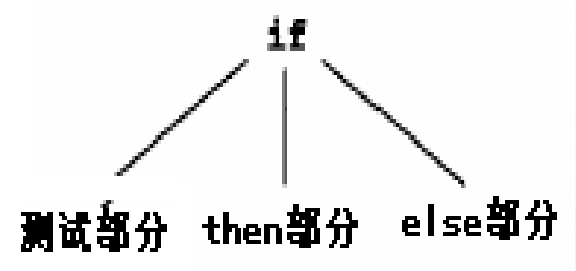
如何为一个程序设计语言设计递归下降语法分析程序

例1：如何为if语句构造相应的语法树？



(1)设计文法：

if-stmt → if (exp) statement |

if (exp) statement else statement

（2）文法改造

if-stmt→if(exp)statement[else statement]

（3）写出递归分析程序

- if语句的EBNF规则：

if-stmt→if(exp)statement[else statement]

分析程序：

```
void  ifStmt( )
{
   match (‘if’) ;
   match (‘(‘) ;
   exp( );
   match (‘)’) ;
   statement( ) ;
   if (TOKEN == ‘else’ ) {
        match (‘else’) ;
        statement( );
    }
} // ifStmt ;
```

（4）语法树生成

按照递归子程序在严格的自顶向下风格构造出if语句的语法树：

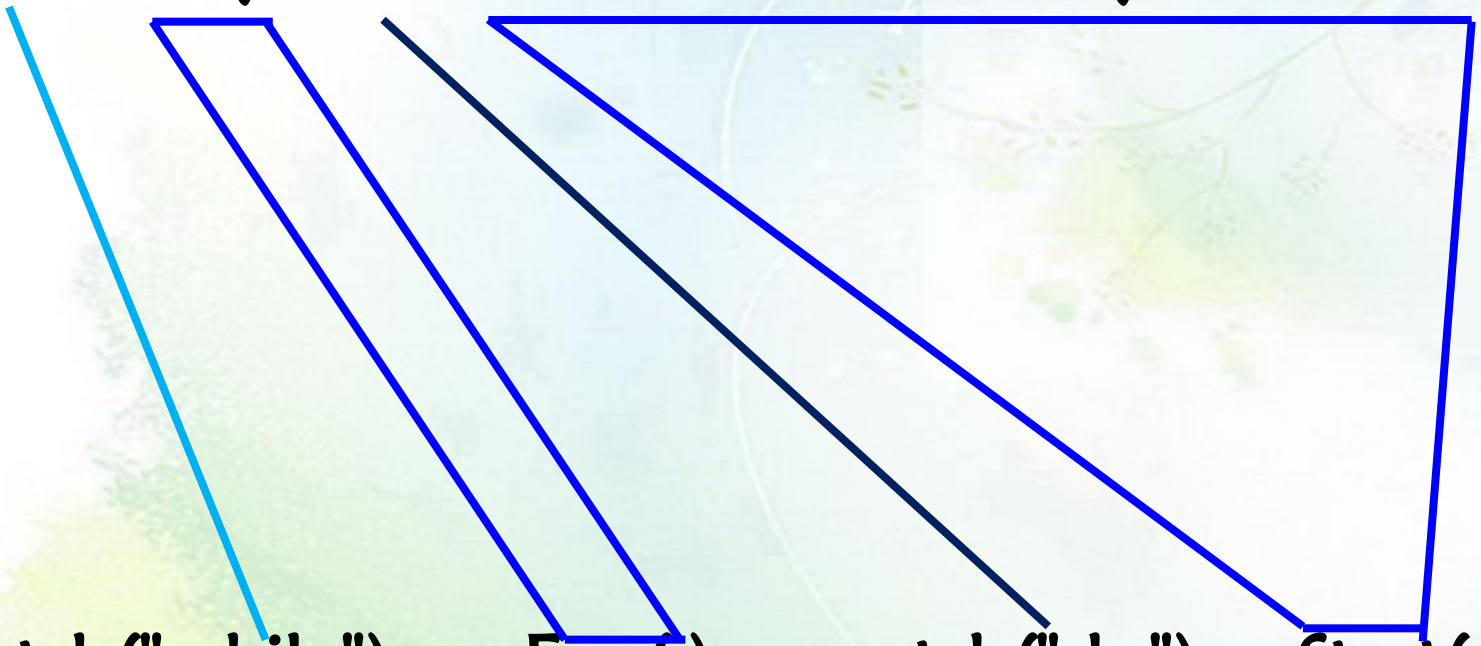
```
syntaxTree ifStatement( )
{   syntaxTree  temp;
     match (“if”) ;
    match (‘(‘) ;
    temp =makeStmtNode(if) ;  //生成新结点
    temp->testChild = exp( ) ;
    match ( ‘)’ ) ;
    temp->thenChild = statement( ) ;
    if (TOKEN == “else” ) {
          match (“else”) ;
          temp->elseChild = statement( ) ;
    } else
          temp->elseChild = NULL ;
} // ifStatement ;
```

例2：stmt→ while Exp do stmt

递归子程序如下：

```
void Stmt( )
{
    match("while") ;
    Exp( ) ;
    match("do") ;
    Stmt( ) ;
}
```

while x>y do if x>z then x:= x+y else x:= y



{ match("while") ; Exp( ); match("do"); Stmt( ); }

Tiny语言的递归下降语法分析程序

- Tiny样本语言：求输入值阶乘TINY语言程序

```
{ Sample program
   in TINY language -
   computes factorial
}
read x; { input an 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
```

TINY 语言的文法规则列表

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

mulop→ * | /

factor→(exp) | number | identifier

EBNF中TINY语言的文法

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **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*]
comparison-op → **<** | **=**
simple-exp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***** | **/**
factor → **(** *exp* **)** | **number** | **identifier**

TINY语言语法分析源程序(云盘)

PARSE.C
PARSE.H
GLOBALS.H

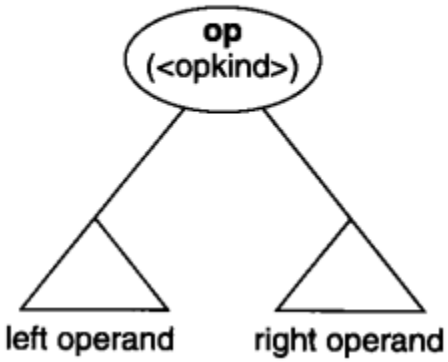
实验三

- TINY语言的递归下降分析程序(自行阅读并运行相关程序(云盘))
- TINY语言的文法　P97 及P136
- 实验三 详细内容和要求见百度云盘文件

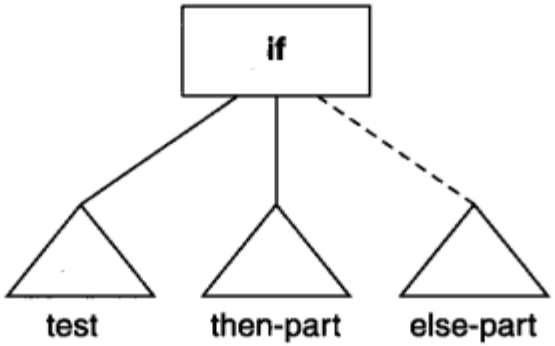
TINY语言语法树的存储结构

云盘参考书 P100

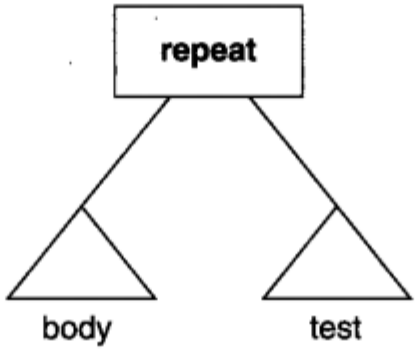
算符表达式有两个孩子，它们表示左操作数表达式和右操作数表达式：



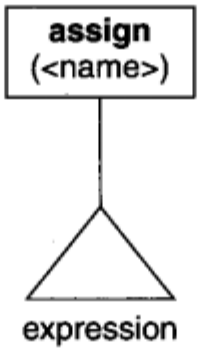
if 语句（带有3个可能的孩子）如下所示：



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）：



其他的语法树结构 云盘参考书 P100

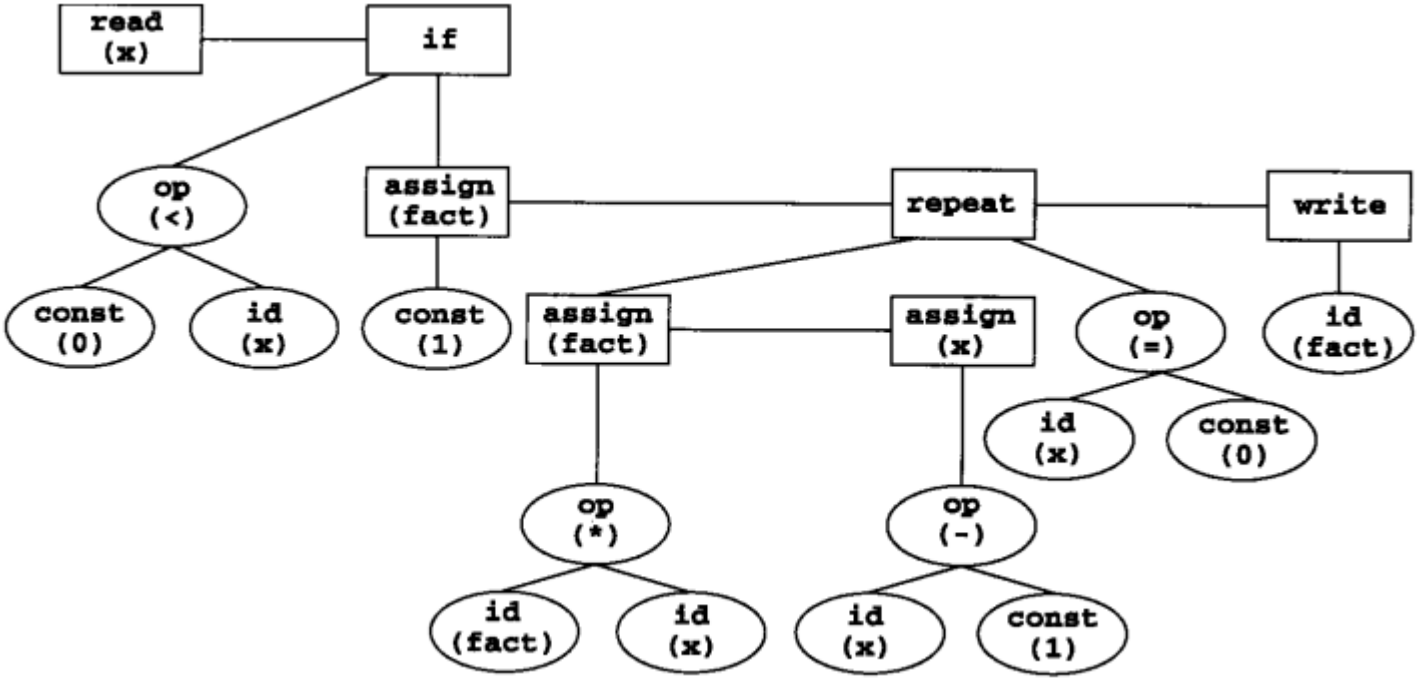
语法树的存储结构（云盘参考书 P98）

程序清单3-2 一个TINY语法树节点的C声明

```
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;
```

- Tiny样本语言：求输入值阶乘TINY语言程序

```
{ Sample program
   in TINY language -
   computes factorial
}
read x; { input an 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
```


read
(x)
if
op
(<)
assign
(fact)
repeat
write
const
(0)
id
(x)
const
(1)
assign
(fact)
assign
(x)
op
(=)
id
(fact)
id
(x)
const
(0)
op
(*)
op
(-)
id
(fact)
id
(x)
id
(x)
const
(1)