**语法分析实验**

**一、实验题目**

编写Tiny语言的语法分析程序。

**二、实验目的**

1 构造Tiny语法分析程序，程序要求能对实现对词法分析程序所提供的Token序列的语法检查和结构分析。

2 在实验的过程中，掌握语法分析的两种方法——自顶向上分析和自底向上分析。

3 使用Yacc工具进行语法分析。

**三、实验要求**

1、独立完成实验

2、由老师现场检查程序代码和运行结果

3、提交代码实验报告：包含程序代码和运行结果截图（电子版本发助教老师QQ邮箱，纸质版打印后教给助教老师）

**四、实验环境**

操作系统：Windows/Linux

开发语言：C/C++

开发工具：Cygwin (在windows平台上运行的UNIX模拟环境)； Yacc工具。

**五、实验内容**

根据下面给出的Tiny语言的语法规则，从由词法分析输出的源程序符号串中识别出各类语法成分，同时进行语法检查。语法分析程序在分析过程中检查符号串是否为该程序的句子。若是则输出该句子的分析树，否则就表示源程序存在语法错误，并报告错误的性质与位置。

1. 采用递归下降分析法来编写程序，实现对给出的两个样例语言（一个是代码完全正确，另外一个代码有少量错误）进行自顶向下的分析。

2. 用Yacc工具对给出的两个样例语言实现自底向上的语法分析。

**六、实验相关知识**

**6.1 Tiny语言的上下文无关文法介绍**

|  |
| --- |
| **Tiny的BNF文法** |
| *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******identifier***  *exp simple-exp comparison-op simple-exp | simple-exp*  *comparison-op < | =*  *simple-exp simple-exp addop term | term*  *addop + | -*  *term term mulop factor | factor*  *multop \* | /*  *factor* ***(****exp****)*** *|****number****|****identifier*** |

|  |
| --- |
| **Tiny的EBNF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*** |

1. Tiny程序是一个语句序列，共有5种语句：if语句，repeat语句，read语句，write语句，write语句和assignment语句。

2. Tiny表达式有两类：（1）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**，（2）包括标准整型算符+、-、\*、/的**算术表达式**。算术表达式是左结合并有通常的优先关系。比较运算是非结合的，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。

3. Tiny中的标识符指的是简单的整型变量。Tiny中没有变量声明，只有通过出现在赋值语句左边来隐式声明一个变量。另外Tiny语言也没有过程或者函数。Tiny语言的语句序列必须包括将语句分隔开来的分号，不能将分号放在语句序列的最后一个语句之后。

**6.2 Tiny语言的语法结构**

语法树节点首先按照它是语句还是表达式来进行分类，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 语句才用到）。语句通过同属域而不是子域来排序，即由父亲到他的孩子的唯一物理连接是到最左孩子的。孩子则在一个标准连接表中自左向右连接到一起，这种连接称作同属连接,用于区别父子连接。

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;

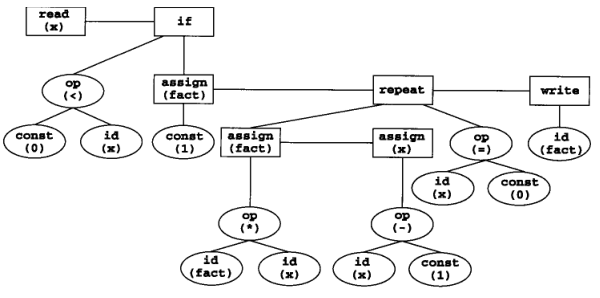
ExpType type; /\* for type checking of exps \*/

} TreeNode;

以Tiny语言的阶乘为例（该代码是正确的）：

|  |
| --- |
| { 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 |

给出Tiny程序的语法树：



**6.3 Yacc工具介绍**

Yacc 代表 Yet Another Compiler Compiler。Yacc的GNU版叫做Bison。它是一种工具，将任何一种编程语言的所有语法翻译成针对此种语言的Yacc语法解析器。它用巴科斯范式(BNF, Backus Naur Form)来书写。按照惯例，Yacc 文件有 .y 后缀。编译行如下调用 Yacc 编译器：

$ yacc <options>

<filename ending with .y>

我们看到 Lex 从输入序列中识别标记。 如果你在查看标记序列，你可能想在这一序列出现时执行某一动作。 这种情况下有效序列的规范称为语法。Yacc 语法文件包括这一语法规范。 它还包含了序列匹配时你想要做的事。

用 Yacc 来创建一个编译器包括四个步骤：

1. 通过在语法文件上运行 Yacc 生成一个解析器。

2. 说明语法：

（1）编写一个 .y 的语法文件（同时说明 C 在这里要进行的动作）。

（2）编写一个词法分析器来处理输入并将标记传递给解析器。 这可以使用 Lex 来完成。

（3）编写一个函数，通过调用 yyparse() 来开始解析。

（4）编写错误处理例程（如 yyerror()）。

3. 编译 Yacc 生成的代码以及其他相关的源文件。

4. 将目标文件链接到适当的可执行解析器库。

如同 Lex 一样, 一个 Yacc 程序也用双百分号分为三段：定义、规则和 子程序。“ 定义”段由一组标记声明和括在“%{”和“%}”之间的C 代码组成。BNF 语法定义放在“规则”段中，而用户子程序添加在“子程序”段中。

|  |
| --- |
| ... 定义 ...  %%  ... 规则 ...  %%  ... 子程序 ... |

YACC 在内部维护着两个堆栈：一个分析栈和一个内容栈。分析栈中保存着终结符和非终结符，并且代表当前剖析状态。内容栈是一个 YYSTYPE 元素的数组，对于分析栈中的每一个元素都保存着一个对应的值。例如，当yylex 返回一个INTEGER标记时，YACC把这个标记移入分析栈。同时，相应的yylval 值将会被移入内容栈中。分析栈和内容栈的内容总是同步的，因此从栈中找到对应于一个标记的值是很容易实现的。

构造一个简单的整型加减法计算器例子：

|  |
| --- |
| %{  int yylex(void);  void yyerror(char \*);  %}  %token INTEGER  %%  program:  program expr '\n' { printf("%d\n", $2); }  |  ;  expr:  INTEGER { $$ = $1; }  | expr '+' expr { $$ = $1 + $3; }  | expr '-' expr { $$ = $1 - $3; }  ;  %%  void yyerror(char \*s) {  fprintf(stderr, "%s\n", s);  return 0;  }  int main(void) {  yyparse();  return 0;  } |

Yacc 语法规则具有以下一般格式：

result: components { /\*

action to be taken in C \*/ }

规则第一条叫 command 规则。其中的左式，或称为非终结符，从最左而开始，后面紧跟着一个自己的克隆。后面跟着的是右式。与规则相应的动作写在后面的花括号中。例如上例中的规则：

|  |
| --- |
| program:  program expr '\n' { printf("%d\n", $2); }  |  ; |

通过利用左递归，我们已经指定一个程序由0 个或更多个表达式构成。每一个表达式由换行结

束。当探测到换行符时，程序就会打印出表达式的结果。当程序应用下面这个规则时

expr: expr '+' expr { $$ = $1 + $3; }

在分析栈中我们其实用左式替代了右式。在本例中，我们弹出“expr '+' expr” 然后压入“expr”。 我们通过弹出三个成员，压入一个成员缩小的堆栈。在我们的 C 代码中可以用通过相对地址访问内容栈中的值，“ $1”代表右式中的第一个成员，“ $2”代表第二个，后面的以此类推。“$$ ”表示缩小后的堆栈的顶部。在上面的动作中，把对应两个表达式的值相加，弹出内容栈中的三个成员，然后把造得到的和压入堆栈中。这样，分析栈和内容栈中的内容依然是同步的。当我们把 INTEGER归约到expr时，数字值开始被输入内容栈中。当 NTEGER 被移分析栈中之后，我们会就应用这条规则 expr: INTEGER { $$ = $1; }。INTEGER标记被弹出分析栈，然后压入一个 expr。 对于内容栈，我们弹出整数值，然后又把它压回去。也可以说，我们什么都没做。事实上，这就是默认动作，不需要专门指定。当遇到换行符时，与expr 相对应的值就会被打印出来。当遇到语法错误时，YACC会调用用户提供的yyerror 函数。如果你需要修改对 yyerror的调用界面，改变YACC包含的外壳文件以适应你的需求。YACC文件中的最后的函数是 main函数。

错误代码

|  |
| --- |
| { Sample program  in TINY language -  computes factorial  }  read x; { input an integer }  if 0 < x { don't compute if x <= 0 }  fact := 1;  repeat  fact := fact \* x;  x := x - 1  until x = 0;  write { output factorial of x }  end |