《编译原理》上机作业（2）

语法分析器

4.2 语法分析器的构造

语法分析器的任务： 分析语言的结构

1. 为句子（表达式）构造语法树；
2. 检查程序（语句）中的语法错误。

主要工作：

1. 设计函数绘图语言的文法，使其适合递归下降分析；
2. 设计语法树的节点，用于存放表达式的语法树；
3. 设计递归下降子程序，分析句子并构造表达式的语法树；
4. 设计测试程序和测试用例，检验分析器是否正确。

4.2.1 函数绘图语言的文法

〈1〉文法

Program → ε | Program Statement SEMICO

Statement → OriginStatment | ScaleStatment
| RotStatment | ForStatment

OriginStatment → ORIGIN IS
L_BRACKET Expression COMMA Expression R_BRACKET

ScaleStatment → SCALE IS
L_BRACKET Expression COMMA Expression R_BRACKET

RotStatment → ROT IS Expression

```
-------- 函数f(t)=t的图形
origin is (200, 300);-- 设置原点的偏移量
rot is pi/6;          -- 设置旋转角度
scale is (2, 1);      -- 设置横、纵坐标比例
for T from 0 to 200 step 1 draw (t, 0);    -- 横坐标
for T from 0 to 180 step 1 draw (0, t);    -- 纵坐标
for T from 0 to 150 step 1 draw (t, t);    -- f(t)=t
```

```
ForStatment → FOR T
    FROM Expression
    TO   Expression
    STEP Expression
    DRAW L_BRACKET Expression COMMA Expression R_BRACKET
Expression
   → Expression PLUS Expression
   |  Expression MINUS Expression
   |  Expression MUL Expression
   |  Expression DIV Expression
   |  PLUS Expression
   |  MINUS Expression
   |  Expression POWER Expression
   |  CONST_ID
   |  T
   |  FUNC L_BRACKET  Expression  R_BRACKET
   |  L_BRACKET  Expression  R_BRACKET
```

〈2〉改写文法为无二义文法

表达式中的运算	结合性	非终结符
PLUS、MINUS（二元）	左结合	Expression
MUL、DIV	左结合	Term
PLUS、MINUS（一元）	右结合	Factor
POWER	右结合	Component
（原子表达式）	无	Atom

```
Expression
    → Expression PLUS Expression
    | Expression MINUS Expression
    | Expression MUL Expression
    | Expression DIV Expression
    | PLUS Expression
    | MINUS Expression
    | Expression POWER Expression
    | CONST_ID
    | T
    | FUNC L_BRACKET  Expression  R_BRACKET
    | L_BRACKET  Expression  R_BRACKET
```

Expression 的改写

Expression对应最低优先级的运算，PLUS和MINUS：

Expression → Expression PLUS Expression
| Expression MINUS Expression

引入Term提高算符的优先级，保留左递归使得算符左结合：

Expression → Expression PLUS Term
| Expression MINUS Term
| Term

Term对应运算MUL和DIV，于是有：

Term → Term MUL Factor
| Term DIV Factor
| Factor

反复改写，最终得到：

无二义的表达式文法

```
Expression → Expression PLUS Term
           | Expression MINUS Term
           | Term
Term       → Term MUL Factor
           | Term DIV Factor
           | Factor
Factor     → PLUS Factor
           | MINUS Factor
           | Component
Component  → Atom POWER Component
           | Atom
Atom       → CONST_ID
           |  T
           |  FUNC L_BRACKET  Expression  R_BRACKET
           |  L_BRACKET  Expression  R_BRACKET
```

Expression	PLUS、MINUS
Term	MUL、DIV
Factor	PLUS、MINUS
Component	POWER
Atom	(原子表达式)

〈3〉 消除左递归和提取左因子

消除program产生式的左递归

Program → Program Statement SEMICO | ε

Program → ε Program'

Program' → Statement SEMICO Program' | ε

Program → Statement SEMICO Program | ε

消除Expression和Term 的左递归

```
Expression → Term  Expression'
Expression'→ PLUS Term Expression'
           | MINUS Term Expression'
           | ε
Term  → Factor Term'
Term' → MUL Factor Term'
      | DIV Factor Term'
      | ε
```

```
Expression →
        Expression PLUS Term
    |   Expression MINUS Term
    |   Term
```

(Factor和Component对应的运算是右结合，故无左递归)

(Component有左因子)

〈4〉改写左结合的产生式为EBNF形式（避免子程序调用）

递归子程序仅要求产生式没有左递归。

Program → Statement SEMICO Program | ε 的子程序：

```
void Program()
{ if (token == NONTOKEN) return;
  Statement(); MathchToken(SEMICO); Program();
}
```

改写为EBNF形式，以减少不必要的子程序调用。

Program → { Statement SEMICO }的子程序：

```
void Program()
{  while (token != NONTOKEN)
    { Statement(); MathchToken(SEMICO); }
}
```

改写Expression产生式：

Expression → Term Expression'

Expression' → PLUS Term Expression'
| MINUS Term Expression' | ε

Expression' → (PLUS|MINUS) Term Expression' | ε

Program → Statement SEMICO Program | ε

Expression' → {(PLUS|MINUS)Term }

Expression → Term {(PLUS|MINUS)Term }

Expression的递归子程序：

```
void Expression()
{     Term();
      while (token==PLUS || token==MINUS)
      { MathchToken(token); Term();}
 }
```

最终表达式的产生式

```
Expression  → Term  { ( PLUS | MINUS) Term }
Term        → Factor { ( MUL | DIV ) Factor }
Factor      → PLUS Factor | MINUS Factor | Component
Component   → Atom [POWER Component]
Atom → CONST_ID
     | T
     | FUNC L_BRACKET Expression R_BRACKET
     | L_BRACKET Expression R_BRACKET
```

函数绘图语言的文法

```
Program   →  { Statement SEMICO }
Statement →   OriginStatment | ScaleStatment
            |  RotStatment     | ForStatment
OriginStatment → ORIGIN IS
    L_BRACKET Expression COMMA Expression R_BRACKET
ScaleStatment  → SCALE IS
    L_BRACKET Expression COMMA Expression R_BRACKET
RotStatment → ROT IS Expression
ForStatment → FOR T
    FROM Expression
    TO   Expression
    STEP Expression
    DRAW L_BRACKET Expression COMMA Expression R_BRACKET
```

```
-------- 函数f(t)=t的图形
origin is (200, 300);-- 设置原点的偏移量
rot is pi/6;          -- 设置旋转角度
scale is (2, 1);      -- 设置横、纵坐标比例
for T from 0 to 150 step 1 draw (t, t);    -- f(t)=t
```

函数绘图语言的文法:表达式的产生式

Expression → Term { (PLUS | MINUS) Term }

Term → Factor { (MUL | DIV) Factor }

Factor → PLUS Factor | MINUS Factor | Component

Component → Atom [POWER Component]

Atom → CONST_ID
| T
| FUNC L_BRACKET Expression R_BRACKET
| L_BRACKET Expression R_BRACKET

主要产生式的递归子程序

```
void Parser(char * SrcFilePtr);
void Program();
void Statement();
void OriginStatement();
void RotStatement();
void ScaleStatement();
void ForStatement();

struct ExprNode * Expression();
struct ExprNode * Term();
struct ExprNode * Factor();
struct ExprNode * Component();
struct ExprNode * Atom();
```

Program → { Statement SEMICO }的子程序：

```
void Program()
{  while (token != NONTOKEN)
     { Statement(); MathchToken(SEMICO); }
}
```

Expression → Term {(PLUS|MINUS)Term }的子程序：

```
void Expression()
{     Term();
      while (token==PLUS || token==MINUS)
       { MathchToken(token); Term();}
 }
```

4.2.2 表达式的语法树

〈1〉语法树的节点

表达式语法树的节点可以设计为以下三类：

1. 叶节点：常数、参数T等。
2. 两个孩子的内部节点：二元运算如Plus、Mul等。

   一元加：+5转化为5；

   一元减：-5转化为0-5。
3. 一个孩子的内部节点：函数调用，如cos(t)等。

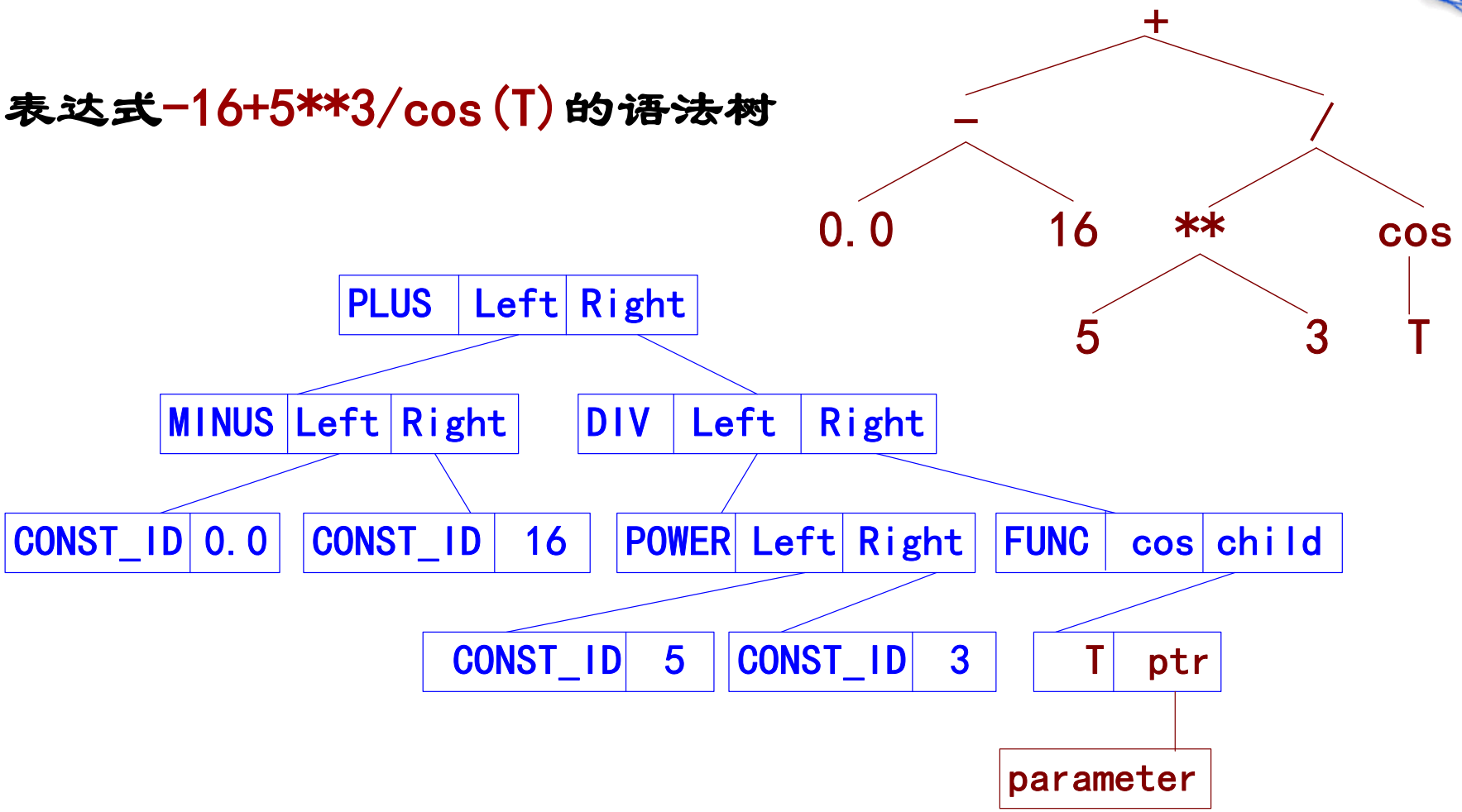
〈2〉节点的数据结构

```
typedef double (* FuncPtr)(double);
struct ExprNode
{  enum Token_Type OpCode;     // 记号种类
   union
   { struct { ExprNode *Left, *Right;
              } CaseOperator; // 二元运算
     struct { ExprNode * Child;
              FuncPtr MathFuncPtr;
              } CaseFunc;      // 函数调用
     double CaseConst;         // 常数，绑定右值
     double * CaseParmPtr;     // 参数T，绑定左值
   } Content;
};
```

〈2〉节点的数据结构

```
typedef double (* FuncPtr)(double);
struct ExprNode
{  enum Token_Type OpCode;    // 记号种类
   union
   {
        struct { ExprNode *Left, *Right;
               } CaseOperator;      // 二元运算
        struct { ExprNode * Child;
                   FuncPtr MathFuncPtr;
               } CaseFunc;    // 函数调用
        double CaseConst;     // 常数，绑定右值
        double * CaseParmPtr; // 参数T，绑定左值
   }Content;
};
```

表达式-16+5**3/cos(T)的语法树



常量绑定的是右值（没有存储空间，值不能被改变）

变量绑定的是左值（有存储空间，值能被改变）

〈3〉 建立语法树的程序框架

```
ExprNode * MakeExprNode( opcode, arg1, arg2,.....)
{ struct ExprNode *ExprPtr = new (struct ExprNode);
  ExprPtr->OpCode = opcode;
  ......
  switch(opcode)
  { case CONST_ID:  // 常数节点
       ExprPtr->Content.CaseConst = arg1; break;
    case T:              // 参数节点
       ExprPtr->Content.CaseParmPtr = &Parameter;
       break;
    case FUNC:           // 函数调用节点
       ExprPtr->Content.CaseFunc.MathFuncPtr = arg1;
       ExprPtr->Content.CaseFunc.Child =(ExprNode *)arg2;
       break;
    default:             // 二元运算节点
       ExprPtr->Content.CaseOperator.Left =(ExprNode *)arg1;
       ExprPtr->Content.CaseOperator.Right=(ExprNode *)arg2;
       break;
  }
}
```

4.2.3 语法分析器的递归下降子程序

〈1〉分析器所需的辅助子程序

```
void FetchToken ();
void MatchToken (enum Token_Type AToken);
void SyntaxError (int case_of);
```

FetchToken源程序：

```
static void FetchToken()
{     token = GetToken(); //调用词法分析器
      if (token.type == ERRTOKEN) SyntaxError(1);
}
```

其中：token是存放记号的全程量；

GetToken()是词法分析器接口；

SyntaxErroe(case_of)是出错处理。

4.2.3 语法分析器的递归下降子程序

〈1〉分析器所需的辅助子程序

```
void FetchToken ();
void MatchToken (enum Token_Type AToken);
void SyntaxError (int case_of);
```

MatchToken源程序：

```
// ------------------------- 匹配记号
static void MatchToken (enum Token_Type The_Token)
{
    if (token.type != The_Token) SyntaxError(2);
    FetchToken();
}
```

〈2〉主要产生式的递归子程序

```
void Parser(char * SrcFilePtr);
void Program();
void Statement();
void OriginStatement();
void RotStatement();
void ScaleStatement();
void ForStatement();

struct ExprNode * Expression();
struct ExprNode * Term();
struct ExprNode * Factor();
struct ExprNode * Component();
struct ExprNode * Atom();
```

Parser的递归子程序

```
void Parser(char * SrcFilePtr)
{
   if(!InitScanner(SrcFilePtr))// 初始化词法分析器
   { printf("Open Source File Error ! \n");
     return;
   }
   FetchToken();       // 获取第一个记号
   Program();          // 递归下降分析
   CloseScanner();     // 关闭词法分析器
} // end of Parser
```

ForStatement的递归子程序

```
ForStatment
  → FOR T FROM Expression TO Expression STEP Expression
    DRAW L_BRACKET Expression COMMA Expression R_BRACKET
```

```
static void ForStatement (void)
{ struct ExprNode *start_ptr, *end_ptr, *step_ptr,
                  *x_ptr, *y_ptr;
  MatchToken (FOR);          MatchToken(T);
  MatchToken (FROM);         start_ptr = Expression();
  MatchToken (TO);           end_ptr   = Expression();
  MatchToken (STEP);         step_ptr  = Expression();
  MatchToken (DRAW);
  MatchToken (L_BRACKET); x_ptr = Expression();
  MatchToken (COMMA);        y_ptr = Expression();
  MatchToken (R_BRACKET);
}
```

Expression的递归子程序

Expression → Term { (PLUS | MINUS) Term }

```
static struct ExprNode *  Expression()
{ struct ExprNode *left, *right;
  Token_Type token_tmp;
  left = Term();
  while (token.type==PLUS || token.type==MINUS)
  {  token_tmp = token.type;
     MatchToken(token_tmp);
     right = Term();
     left = MakeExprNode(token_tmp,left,right);
  }
  return left;
}
```

根据Expression的递归子程序，不难写出其它表达式的递归子程序。

4.2.4 语法分析器的测试

〈1〉测试主程序与测试辅助子程序

a) 测试主程序

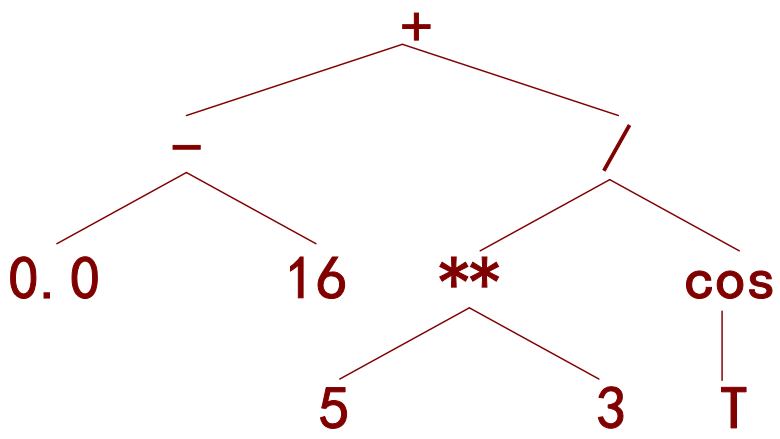
```
#include <stdio.h>
extern void Parser(char * SrcFilePtr);
void main(int argc, char *argv[])
{ if(argc<2){printf( "Input Source!\n" ); return; }
  Parser(argv[1]);
}
```

b) 打印语法树的子程序

```
void PrintSyntaxTree(struct ExprNode *root, int indent);
```

从root开始，对语法树进行深度优先的先序遍历，并且根据缩进值indent将当前被遍历的节点打印在适当的位置上。

-16+5**3/cos(T)的语法树：




```
+
    -
        0.000000
        16.000000
    /
        **
            5.000000
            3.000000
        402da4
        T
```

〈2〉 测试语句的嵌入与测试结果

a) 测试语句的加入：

1. 子程序入口与出口加入“enter”和“exit”
2. 终结符匹配后加入“mathctoken ***”
3. 表达式（Expression）构造后，打印语法树

b) 语法分析器应测试的内容：

1. 重要语言结构：各类语句、各种算术表达式
2. 典型的语法错误：语句错误、算术表达式错误等

c） 被测试源程序举例：

```
-16+5**3/cos(T)                -- 不是语句
rot is -16+5**3/cos(T)         -- 行尾少分号
rot is -16+5**3/cos(T);        -- 正确语句

-------- 函数图形
origin is (350, 200);          -- 设置原点的偏移量
rot is pi/6;                   -- 设置旋转角度
scale is (2, 1);               -- 设置横、纵坐标比例
for t from -100 to 100 step 1 draw (t, 0);    -- 横坐标
for t from -100 to 100 step 1 draw (0, t);    -- 纵坐标
scale is (200, 100);           -- 设置横、纵坐标比例
for t from 0 to 2*pi step pi/50 draw (cos(t),sin(t));
```

d） 测试结果（看程序运行）

结　束

改写Program产生式为EBNF形式

对于产生式：

Program → Statement SEMICO Program | ε

按其不同的右部候选项展开，会得到下述序列：

ε，

Statement SEMICO,

Statement SEMICO Statement SEMICO, ...

即“Statement SEMICO”被重复0或若干次，于是有：

Program → { Statement SEMICO }

返回

〈3〉建立语法树的递归子程序（78页）

```
#include <stdarg.h>
double Parameter;   // 参数
struct ExprNode * MakeExprNode(enum Token_Type
opcode, ...)
{ struct ExprNode *ExprPtr = new (struct ExprNode);
  va_list ArgPtr;
  ExprPtr->OpCode = opcode;
  va_start(ArgPtr, opcode);
  switch(opcode)
  { case CONST_ID:    // 常数节点
        ExprPtr->Content.CaseConst
            =(double)va_arg(ArgPtr,double);
        break;
    case T:                // 参数节点
        ExprPtr->Content.CaseParmPtr=&Parameter;
        break;
```

〈3〉 建立语法树的递归子程序（续）

```
    case FUNC:// 函数调用节点
        ExprPtr->Content.CaseFunc.MathFuncPtr
                = (FuncPtr)va_arg(ArgPtr, FuncPtr);
        ExprPtr->Content.CaseFunc.Child
               =(struct ExprNode *)va_arg(ArgPtr,struct ExprNode *);
        break;
     default:  // 二元运算节点
        ExprPtr->Content.CaseOperator.Left
               =(struct ExprNode *)va_arg(ArgPtr,struct ExprNode *);
        ExprPtr->Content.CaseOperator.Right
               =(struct ExprNode *)va_arg(ArgPtr,struct ExprNode *);
        break;
  }     // end of switch
  va_end(ArgPtr);
  return ExprPtr;
} // end of MakeExprNode
```