编译简介

文法规则

用“ ::= ”符号，表示“ 由……组成 ”。文法规则一般形如：

<句子> ::= <主语> <谓语>

<谓语> ::= <动词> <宾语>

表示“ 句子由主语和谓语构成 ”，“ 谓语由动词和宾语构成 ”。这种构成是按顺序的。

对于语法中的大写字母，或者用尖括号括起来的部分，称为“ 语法项 ”或者“ 非终结符号 ”。非终结符号可以出现在文法规则的左部或者右部。

对于语法中的小写字母，或者用单引号引起来的部分，表示具体的字符，称为“ 终结符号 ”。终结符号只能出现在文法规则的右部，无法再向下推导。最终待匹配的字符串只由终结符号构成。

文法G[S]的含义是，文法G的“ 起始符号 ”是S□相当于语法树的根节点。起始符号规定为一个非终结符号。

文法规则中有一些特殊符号。

符号ε□希腊字母epsilon□表示空串，即长度为0的字符串，就是什么都没有。空串的概念类似于空集，但是空串是串，仍旧作为元素来看待，与空集概念有区别。

符号|：表示“ 或 ”。例如：

A ::= B | C

表示A由B构成，或者由C构成。这里的B和C称为文法项A的候选式。

符号{}：大括号里的内容，表示可以重复0次到重复任意有限次。不能重复无限次，因为字符串永远有限长。例如：

A ::= {a}

表示A可以匹配ε□a□aa□aaa□.....□所有只由a构成的字符串。

如果要表示这些特殊符号，一般会用单引号引起来，或者用转义字符'\'去转义。

语法分析

语法分析是编译的一个环节，可以检查输入的字符串是否符合文法规则。

语法分析的思维模式总共分为两种：自顶向下的分析、自底向上的分析，它们按照遍历语法树的顺序来定义。

递归下降属于自顶向下的分析，优先爬升属于自底向上的分析。

递归下降

递归下降是一种语法分析的设计方法。

能够递归下降的文法，需要满足3个条件：

1.没有左递归。例如文法

A ::= a | Ab

无法使用递归下降，必须改写为与之等价的

A ::= a{b}

才能使用递归下降。

2.候选式首符号不相交。例如文法：

A ::= ab | ac

无法使用递归下降。必须改写为与之等价的

A ::= aB

B ::= b | c

才能使用递归下降。

3.如果候选式可以为ε则候选式的首符号与该语法项的后继符号也不相交。例如文法：

S ::= Aa

A ::= ε | ab

无法使用递归下降。必须改写为与之等价的

S ::= aB

B ::= ε | ab

才能使用递归下降。

递归下降的设计方法是：

对每一个左部的语法项，设计一个函数。

进入函数时，根据读入的首符号，来确定进入哪个候选式分支。

如果遇到ε先预读一个符号，如果判断为语法项的后继符号，则进入该分支，于是退回一个符号并返回。

如果遇到大括号，通过while循环来实现循环0次与无数次的目的。在while的判断处要读入字符检查是否进入循环，因此在while结束后要退回一个字符。

示例：括号匹配的文法G[S]为

S ::= A

A ::= ε | '(' A ')' A | '[' A ']' A | '{' A '}' A

于是设计的程序为：

```
int S() {
    int ans = A();//括号匹配
    if (ans == 0) {
        return 0;
    }
    char c = getchar();//匹配完应该读完
    if (c != EOF) {
        return 0;
    }
    return 1;
}

int A() {
    char c = getchar();
    if (c == ')' || c == ']' || c == '}' || c == EOF) {
        ungetc(c, stdin);//与getchar相反，向读入中退回一个字符
        return 1;
    } else if (c == '(') {
        int temp = A();
        if (temp == 0)//调用匹配失败
        {
            return 0;
        }
        c = getchar();
        if (c != ')') {
            return 0;
        }
        temp = A();
        if (temp == 0) {
            return 0;
        }
        return 1;
    } else if (c == '[') {
        int temp = A();
        if (temp == 0) {
            return 0;
        }
        c = getchar();
        if (c != ']') {
            return 0;
        }
        temp = A();
        if (temp == 0) {
            return 0;
        }
        return 1;
    } else if (c == '{') {
        int temp = A();
        if (temp == 0) {
            return 0;
        }
```

```
        c = getchar();
        if (c != '}') {
            return 0;
        }
        temp = A();
        if (temp == 0) {
            return 0;
        }
        return 1;
    }
}
```

算符优先

文法可能有二义性。例如：

A ::= 'x' | A '+' A | A '*' A

就具有二义性。因为对于字符串x+x*x□有两种解读方式：先运算x*x□或者先运算x+x□

解决二义性有两种办法：

1.改写为无二义性的文法。比如：

A ::= B '+' A

B ::= C '*' B

C ::= x

没有二义性。

2.对算符引入优先级。这种办法只适用于算符优先文法。

例如对上面的文法，强行定义算符'+'的优先级是1，算符'*'的优先级是2，于是就人为地规定了算符的运算顺序：先运算'*'，再运算'+'。

算符优先文法□Operator Precedence Gramma, OPG□是一种特殊的文法。它要求：在文法规则的右部，每两个终结符号都不相邻。例如上面给出的

A ::= 'x' | A '+' A | A '*' A

就是算符优先文法。算符优先文法的每个终结符号都称为算符，右部中连接两个非终结符号的终结符号称为二元算符。。

算符优先文法常常具有二义性，这时为了方便，常常需要为二元算符定义优先级来消除二义性。

优先爬升

几乎所有无二义性的与上下文无关的文法，都能够改写为可以递归下降的文法。然而，适用于优先爬升的文法，只有引入了算符优先级的算符优先文法。

优先爬升（Precedence Climbing）的方法，特别适合处理表达式问题。

优先爬升算法的一个示例如下：

```
struct expression parse() {
    struct expression lhs = parse_primary();
    return parse_opg(lhs, 0);
}

struct expression parse_opg(struct expression lhs, int precedence) {
    char peek=getchar();
    while (is_binary_op(peek)&&prec(peek)>=precedence) {
        char op=peek;
        struct expression rhs = parse_primary();
        peek=getchar();
        while (is_binary_op(peek) && ( prec(peek) > prec(op) || (
is_right_assoc(peek) && prec(peek) == prec(op) ))) {
            ungetc(peek,stdin);
            rhs = parse_opg(rhs, prec(peek));
            peek=getchar();
        }
        lhs = combine(lhs, op, rhs);
    }
    ungetc(peek,stdin);
    return lhs;
}
```

其中：

函数is_binary_op(char peek)判断是否二元算符。

函数is_right_assoc(peek)判断该二元算符是否具有右结合性，即先计算右边，再计算左边。

函数prec(char peek)计算算符优先级。优先级最低为1，规定先运算高优先级，再运算低优先级。

函数combine(struct expression lhs, char op, struct expression rhs)将二元表达式的各部分组合。

函数parse_primary()解析一元表达式。

上面的优先爬升算法仅仅解决了最普通的算符优先文法，即只有二元算符的算符优先文法。

一般认为，前置一元算符的优先级高于二元算符。如果我们对函数parse_primary()稍加改造，就可以处理具有前置一元算符与括号的算符优先文法。一个示例：

```
struct expr parse_primary()
{
    char next=getchar();
    if(next=='(')
    {
        struct expression temp=parse();//括号里允许低优先级的二元算符
        next=getchar();//右括号
        return temp;
    }
```

```
    else if(next=='-')//假设'-'是一个前置一元运算符
    {
        struct expression temp=parse_primary();//一般前置一元算符的优先级高于二元算符
        return -temp;//这里对表达式的结果取负的过程，需要自己写
    }
    else//普通标识符
    {
        return expression(next);//这里利用temp构造一个表达式的过程，需要自己写
    }
}
```

一般认为，后置一元算符的优先级也高于二元算符。如果对函数parse_opg(struct expression lhs, int precedence)中，每个while前的peek=getchar()语句后面再插入一个while语句，还可以处理后置一元算符。例如：

```
    peek=getchar();
    while(peek=='.')//假设'.'是一个后置一元算符，并且优先级也高于二元算符
    {
        struct expression lhs=dot(lhs);//这里处理点运算符，需要自己写
        peek=getchar();
    }
    while (is_binary_op(peek)&&prec(peek)>=precedence)
```

根据该代码段插入的位置不同，可以将lhs对应写成rhs

如果对函数parse_primary()的普通标识符分支再加一个预读，还可以处理函数调用。例如，假设函数调用的语法是标识符后面接一个括号：

```
    else//普通标识符
    {
        char temp=getchar();
        if (temp=='(')
        {
            return function(next);//这里处理整个函数调用，需要自己写
        }
        else
        {
            ungetc(temp,stdin);
            return expression(next);//这里利用temp构造一个表达式的过程，需要自己写
        }
    }
```