编译器构造实验-2

19335206 韦媛馨

2022-05-03

本实验GitHub链接：https://github.com/Atopos-309/Compilation-principle

1 实验描述

算术表达式语法分析器的设计与实现

2 实验要求

使用LL(1)分析法和LR分析法设计实现算术表达式的语法分析器

算数表达式至少支持加减乘除以及括号操作，即（+，-，*，/，（））

3 文法

```
E->E+T|E-T|T
T->T*F|T/F|F
F->i|(E)
```

4 LL(1)分析法

4.1 实验思路

LL(1)文法的条件如下：

1. 文法不含左递归
2. 对文法中每一个非终结符A的各个产生式的FIRST集合两两不相交。
3. 对文法中每一个非终结符A，若存在某个FIRST集合包含ε，则： First(A)∩ Follow(A)=ø

LL(1)分析法的实验思路如下：

1. 构造文法
2. 改造文法：消除二义性、消除左递归、消除回溯
3. 求每个变量的FIRST集和FOLLOW集合
4. 检查是不是LL(1)文法，若是，构造预测分析表
5. 实现表驱动的预测分析算法

4.2 文法消除左递归

```
E->TA A->+TA A->-TA A->e
T->FB B->*FB B->/FB B->e
F->i F->(E)
其中：e: epsilon
```

4.3 First集、Follow集的计算

```
First(E)=First(A)={+,-,e}
First(T)=First(B)={*,/,e}
First(F)={i,(}

Follow(A)=Follow(E)={$,)}
Follow(B)=Follow(T)={+,-,$,)}
Follow(F)={*,/,+,-,$}
```

根据LL(1)文法的判断条件，可以判断出该文法符合LL(1)文法，可以构造预测分析表。

4.4 构造预测分析表

本次实验中错误恢复使用的恐慌模式，也就是忽略输入的一些符号，直到输入中出现合法的词法单元。

构造预测分析表的算法思想为：

1. 遍历所有产生式，对每一个产生式`A->α`：
    1. 如果右部的第一个字符是非终结符，遍历它的First集，更新预测分析表，即`Table[A][x] = A->α`（`x`为First集字符）
    2. 如果右部的第一个字符是空串，遍历左部的Follow集，更新预测分析表，即`Table[A][y] = A->α`（`y`为Follow集字符）
2. 写入同步信息，将`Follow(A)`中的所有终结符放入非终结符A的同步记号集合，`synch`表示根据相应非终结符的Follow集得到的同步词法单元

根据上述算法思想，关键编程实现如下：

```
// 遍历G的每个产生式
for (auto itG = G.begin(), itFirst = First.begin(); itG != G.end() && itFirst != First.end(); ++itG, ++itFirst){
    // 非终结符下标转换
    int x = index.at(*(itG->begin()));
    for (auto first = itFirst->begin(); first != itFirst->end(); ++first){
        //如果右部的第一个字符是非终结符，遍历它的First集，更新预测分析表
        if (*first != 'e'){
            int y = index.at(*first);
            table.at(x).at(y) = *itG;
        }
        //如果右部的第一个字符是空串，遍历左部的Follow集，更新预测分析表
        else{
            for (auto follow = Follow.at(x).begin(); follow != Follow.at(x).end(); ++follow){
                int y = index.at(*follow);
                table.at(x).at(y) = *itG;
            }
        }
    }
}
// 写入同步信息
for (string::size_type i = 0; i < nonTerminal.length(); ++i){
    int x = index.at(nonTerminal.at(i));
    for (vector<string>::size_type j = 0; j < Follow.at(i).length(); ++j){
        int y = index.at(Follow.at(i).at(j));
        if (table.at(x).at(y).empty())
            table[x][y] = "synch";
    }
}
```

根据本次实验所定义的文法，运行程序后输出预测分析表如下：

```
==============================LR(0) Analysis Table============================

        i       +       -       *       /       (       )       $
E       E->TA                                   E->TA   synch   synch
A               A->+TA  A->-TA                          A->e    A->e
T       T->FB   synch   synch                   T->FB   synch   synch
B               B->e    B->e    B->*FB  B->/FB          B->e    B->e
F       F->i    synch   synch   synch   synch   F->(E)  synch   synch
```

可见，该预测分析表没有多重定义的表项，因此可以进一步检验该文法符合LL(1)文法。

4.5 LL(1)分析过程

在上一步已经构造出了预测分析表，该分析表的使用方法为：

1. 如果`Table[A][a]`是空，表示检测到错误，根据恐慌模式，忽略输入符号`a`
2. 如果`Table[A][a]`是`synch`，则弹出栈顶的非终结符A，试图继续分析后面的语法成分
3. 如果栈顶的终结符和输入符号不匹配，则弹出栈顶的终结符

实现LL(1)分析过程的核心代码如下：

```
do{
    // 输出当前栈和当前剩余输入
    string str1(analyStack.begin(), analyStack.end());
    string str2(ip, s.end());
    cout << left << setw(wid + 10) << str1 << right << setw(wid) << str2 << "
";

    // 栈顶和当前输入符号
    top = analyStack.back();
    cur = *ip;

    // 栈顶是终结符号或$
    if (terminal.find(top) != terminal.npos || top == '$'){
        if (top == cur){
            analyStack.pop_back();
            ++ip;
            cout << endl;
        }
        else{
            cout << "出错! 不匹配,弹出" << top << endl;
            analyStack.pop_back();
        }
    }
    // 栈顶非终结符
    else{
        //坐标转换
        int x = index.at(top);
        int y;
        try{
            y = index.at(cur);
        }
        catch (out_of_range){
            cout << "输入字符非法! " << endl;
            break;
```

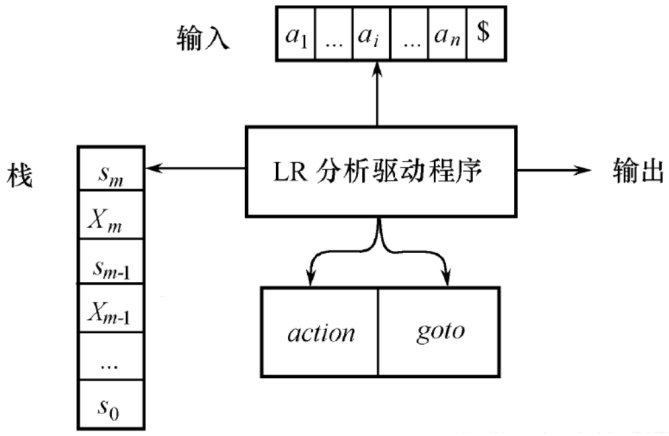
```
        }

        // 产生式
        string production = table.at(x).at(y);
        // 产生式非空
        if (!production.empty()){
            if (production == "synch"){    //同步
                cout << "出错!synch,弹出" << top << endl;
                analyStack.pop_back();
            }
            else{    //正常分析
                analyStack.pop_back();
                string expr(production.begin() + 3, production.end());
                if (expr == "e")  expr = "";
                // 逆序压栈
                for (auto iter = expr.rbegin(); iter != expr.rend(); ++iter)
                    analyStack.push_back(*iter);
                // 输出产生式
                cout << production << endl;
            }
        }
        else {   //表项空白
            cout << "出错! 空白,跳过" << *ip << endl;
            ++ip;
        }
    }
}while (top != '$');
```

5 LR分析法

5.1 实验思路

如果某一文法能够构造一张LR分析表，使得表中每一个元素至多只有一种明确动作，则该文法称为LR文法。LR分析器是一种由下而上（bottom-up）的上下文无关语法分析器。LR分析器的模型如图所示，它包括输入、输出、栈、驱动程序和含动作和转移两部分的分析表。



驱动程序对所有的LR分析方法都一样，不同的分析方法构造的分析表不同。分析程序每次从输入缓冲区读一个符号，它使用栈存储形式为$s_0...X_{m-1}s_{m-1}X_ms_m$的串，$s_m$在栈顶。$X_i$是文法符号，$s_i$是叫做状态的符号，状态符号概括了栈中它下面部分所含的信息。栈顶的状态符号和当前的输入符号用来检索分析表，以决定移进—归约分析的动作。

LR0分析的实验思路为：

1. 文法设计及扩展文法
2. 根据LR分析法的步骤在每个产生式的右部适当位置添加一个圆点构成项目
3. 构造项目规范簇
4. 画出项目规范簇的DFA
5. LR分析表的构造
6. 构造词法分析器
7. 模拟LR分析器的工作过程进行语法分析
8. 输出分析过程

5.2 文法设计及扩展文法

实验中定义的文法为

```
E->E+T|E-T|T
T->T*F|T/F|F
F->i|(E)
```

将其扩展如下：

```
E'->E
E->E+T
E->E-T
E->T
T->T*F
T->T/F
T->F
F->(E)
F->i
```

5.3 写出文法项目

```
(0)E'->·E
(1)E->·E+T   (2)E->E·+T   (3)E->E+·T   (4)E->E+T·
(5)E->·E-T   (6)E->E·-T   (7)E->E-·T   (8)E->E-T·
(9)E->·T     (10)E->T·
(11)T->·T*F  (12)T->T·*F  (13)T->T*·F  (14)T->T*F·
(15)T->·T/F  (16)T->T·/F  (17)T->T/·F  (18)T->T/F·
(19)T->·F    (20)T->F·
(21)F->·(E)  (22)F->(·E)  (23)F->(E·)  (24)F->(E)·
(25)F->·i    (26)F->i·
```

5.4 构造LR分析表(即Action表和Goto表)

构造项目规范簇并画出其DFA，之后构造LR分析表如下：

```
==============================LR(0) Analysis Table============================

-       +       -       *       /       i       (       )       #       E       T       F
0                                       S4      S5                      1       2       3
1       S6      S7                                                      acc
2       r3      r3      S8      S9                      r3      r3
3       r6      r6      r6      r6                      r6      r6
5                                       S4      S5                      10      2       3
6                                       S4      S5                              11      3
7                                       S4      S5                              12      3
8                                       S4      S5                                      13
9                                       S4      S5                                      14
10      S6      S7                                      S15
11      r1      r1      S8      S9                      r1      r1
12      r2      r2      S8      S9                      r2      r2
13      r4      r4      r4      r4                      r4      r4
14      r5      r5      r5      r5                      r5      r5
15      r7      r7      r7      r7                      r7      r7
```

5.5 LR(0)分析过程

算法　LR 分析算法。

输入：文法 G 的 LR 语法分析表和输入串 w。

输出：如果 w 属于 $L(G)$，则输出 w 的自底向上分析，否则报错。

方法：首先，把初始状态 s_0 放在语法分析器栈顶，把 $w\$$ 放在输入缓冲区；然后，语法分析器执行程序，直到遇见接受或出错动作为止。

```
令 ip 指向w$ 的第一个符号；
repeat forever begin
    令s 是栈顶的状态，a 是ip 所指向的符号；
    if action[s, a] = "移动状态s' 进栈" then begin
        把a和s' 依次压入栈顶；
        让 ip 指向下一个输入符号
    end
    else if action[s, a] = "按文法产生式A→β归约" then begin
        从栈顶弹出 2*|β| 个符号；
        令s'是现在的栈顶状态；
        把A和goto[s', A]依次压入栈；
        输出产生式A→β
    end
    end if action [s, a] = "接受" then
        return
    else error()
end
```

LR 分析程序

LR0分析的关键代码如下，为了精简展示，省略了部分打印信息和相对重要性较低的代码，完整代码可见`LR0.hpp`：

```
while(LR<=len)
    x <- Status.top(); //状态栈栈顶
    y <- findL(str[LR]); //待判断串串首
    l <- strlen(LR0[x][y]);//当前Ri或Si的长度

    if LR0[x][y][0]=='a' then
        acc;
        return

```

```
    else if LR0[x][y][0]=='S' then //Si
        t <- calculate(l,LR0[x][y]);//整数
        push t into Status
        push str[LR] into Symbol
        push num_of_steps into status_arr
        LR++;

    else if LR0[x][y][0]=='r' then  //ri,退栈，ACTION和GOTO
        t <- calculate(l,LR0[x][y]);
        g <- del[t];
        while(g--)
            Status.pop
            Symbol.pop
        g <- del[t];
        while(g>0)
            fix Status_arr
            g--
        push head[t] into Symbol
        symbol_arr[symbol_ind] = head[t];
        x <_ Status.top();
        y <- findL(Symbol.top());
        Determine the num_of_arr corresponding to the Symbol stack
        push t into Status

    else
        t <- LR0[x][y][0]-'0';
        push t into Status
        status_arr[status_ind] = LR0[x][y][0];
        symbol_arr[++symbol_ind] = 'E';
        LR++;
```

6 实验结果

6.1 运行方式说明

cd到当前文件夹，确保`main.cpp`,`LL1.hpp`,`LR0.hpp`在同一目录下。编译生成可执行文件`parser.exe`：

```
g++ -o parser main.cpp
```

运行：

```
./parser
```

根据提示输入算术表达式：

```
Please enter an arithmetic expression:
i+i*i
```

回车之后程序开始对该算术表达式进行语法分析，使用LL1和LR0两种不同的方法，分析结果输出到控制台中。

6.2 LL(1)分析法

1. 对于算术表达式`i*(i+i)`，文法分析输出如下：

```
============================Parsing syntax using LL(1)============================
Stack                Input    Output
$E                 i*(i+i)$    E->TA
$AT                i*(i+i)$    T->FB
$ABF               i*(i+i)$    F->i
$ABi               i*(i+i)$
$AB                 *(i+i)$    B->*FB
$ABF*               *(i+i)$
$ABF                 (i+i)$    F->(E)
$AB)E(               (i+i)$
$AB)E                 i+i)$    E->TA
$AB)AT                i+i)$    T->FB
$AB)ABF               i+i)$    F->i
$AB)ABi               i+i)$
$AB)AB                 +i)$    B->e
$AB)A                  +i)$    A->+TA
$AB)AT+                +i)$
$AB)AT                  i)$    T->FB
$AB)ABF                 i)$    F->i
$AB)ABi                 i)$
$AB)AB                   )$    B->e
$AB)A                    )$    A->e
$AB)                     )$
$AB                       $    B->e
$A                        $    A->e
$                         $
```

最后栈中只有`$`，说明表达式正确。

2. 对于算术表达式`(i-i)*i+i/i`，文法分析输出如下：

```
============================Parsing syntax using LL(1)============================
Stack                        Input    Output
$E                       (i-i)*i+i/i$    E->TA
$AT                      (i-i)*i+i/i$    T->FB
$ABF                     (i-i)*i+i/i$    F->(E)
$AB)E(                   (i-i)*i+i/i$
$AB)E                     i-i)*i+i/i$    E->TA
$AB)AT                    i-i)*i+i/i$    T->FB
$AB)ABF                   i-i)*i+i/i$    F->i
$AB)ABi                   i-i)*i+i/i$
$AB)AB                     -i)*i+i/i$    B->e
$AB)A                      -i)*i+i/i$    A->-TA
$AB)AT-                    -i)*i+i/i$
$AB)AT                      i)*i+i/i$    T->FB
$AB)ABF                     i)*i+i/i$    F->i
$AB)ABi                     i)*i+i/i$
$AB)AB                       )*i+i/i$    B->e
$AB)A                        )*i+i/i$    A->e
$AB)                         )*i+i/i$
$AB                           *i+i/i$    B->*FB
$ABF*                         *i+i/i$
$ABF                           i+i/i$    F->i
$ABi                           i+i/i$
$AB                             +i/i$    B->e
$A                              +i/i$    A->+TA
$AT+                            +i/i$
$AT                              i/i$    T->FB
$ABF                             i/i$    F->i
$ABi                             i/i$
$AB                               /i$    B->/FB
$ABF/                             /i$
$ABF                               i$    F->i
$ABi                               i$
$AB                                 $    B->e
$A                                  $    A->e
$                                   $
```

最后栈中只有`$`，说明表达式正确。

3. 对于算术表达式`i++i*i`，文法分析输出如下：

```
============================Parsing syntax using LL(1)============================
Stack              Input    Output
$E               i++i*i$    E->TA
$AT              i++i*i$    T->FB
$ABF             i++i*i$    F->i
$ABi             i++i*i$
$AB               ++i*i$    B->e
$A                ++i*i$    A->+TA
$AT+              ++i*i$
$AT                +i*i$    出错!synch,弹出T
$A                 +i*i$    A->+TA
$AT+               +i*i$
$AT                 i*i$    T->FB
$ABF                i*i$    F->i
$ABi                i*i$
$AB                  *i$    B->*FB
$ABF*                *i$
$ABF                  i$    F->i
$ABi                  i$
$AB                   $    B->e
$A                    $    A->e
$                     $
```

通过分析过程可以看到，在中间遇到`++`时，根据恐慌模式的错误恢复策略，将输入串的第二个`+`弹出，并继续　分析。

4. 对于算术表达式`i-i/i+i^i`，文法分析输出如下：

```
============================Parsing syntax using LL(1)============================
Stack                   Input    Output
$E                  i-i/i+i^i$    E->TA
$AT                 i-i/i+i^i$    T->FB
$ABF                i-i/i+i^i$    F->i
$ABi                i-i/i+i^i$
$AB                  -i/i+i^i$    B->e
$A                   -i/i+i^i$    A->-TA
$AT-                 -i/i+i^i$
$AT                   i/i+i^i$    T->FB
$ABF                  i/i+i^i$    F->i
$ABi                  i/i+i^i$
$AB                    /i+i^i$    B->/FB
$ABF/                  /i+i^i$
$ABF                    i+i^i$    F->i
$ABi                    i+i^i$
$AB                      +i^i$    B->e
$A                       +i^i$    A->+TA
$AT+                     +i^i$
$AT                       i^i$    T->FB
$ABF                      i^i$    F->i
$ABi                      i^i$
$AB                        ^i$    输入字符非法!
```

可以看到，对于`^`，它并不在我们所设计的文法中。因此输入串遇到`^`时会提示“输入字符非法”的错误信息，分析终止。

可见，实验中所设计的LL1文法分析器可以正确进行文法分析并进行错误处理。

6.3 LR(0)分析法

1. 对于算术表达式`i*(i+i)`，文法分析输出如下：

```
============================Parsing syntax using LR(0)============================
Step      Status Stack    Symbol Stack    Input    ACTION    GOTO
(1)             0               #         i*(i+i)#  S4
(2)             04              #i        *(i+i)#   r8        3
(3)             03              #F        *(i+i)#   r6        2
(4)             02              #T        *(i+i)#   S8
(5)             028             #T*       (i+i)#    S5
(6)             0285            #T*(      i+i)#     S4
(7)             02854           #T*(i     +i)#      r8        3
(8)             02853           #T*(F     +i)#      r6        2
(9)             02852           #T*(T     +i)#      r3        10
(10)            0285(10)        #T*(E     +i)#      S6
(11)            0285(10)6       #T*(E+    i)#       S4
(12)            0285(10)64      #T*(E+i   )#        r8        3
(13)            0285(10)63      #T*(E+F   )#        r6        11
(14)            0285(10)6(11)   #T*(E+T   )#        r1        10
(15)            0285(10)        #T*(E     )#        S15
(16)            0285(10)(15)    #T*(E)    #         r7        13
(17)            028(13)         #T*F      #         r4        2
(18)            02              #T        #         r3        1
(19)            01              #E        #         acc
```

`ACTION`列最后显示`acc`，说明该表达式正确，成功被接收。

2. 对于算术表达式`(i-i)*i+i/i`，文法分析输出如下：

```
============================Parsing syntax using LR(0)============================
Step      Status Stack    Symbol Stack    Input    ACTION    GOTO
(1)             0               #         (i-i)*i+i/i#S5
(2)             05              #(        i-i)*i+i/i#S4
(3)             054             #(i       -i)*i+i/i#r8        3
(4)             053             #(F       -i)*i+i/i#r6        2
(5)             052             #(T       -i)*i+i/i#r3        10
(6)             05(10)          #(E       -i)*i+i/i#S7
(7)             05(10)7         #(E-      i)*i+i/i# S4
(8)             05(10)74        #(E-i     )*i+i/i#  r8        3
(9)             05(10)73        #(E-F     )*i+i/i#  r6        12
(10)            05(10)7(12)     #(E-T     )*i+i/i#  r2        10
(11)            05(10)          #(E       )*i+i/i#  S15
(12)            05(10)(15)      #(E)      *i+i/i#   r7        3
(13)            03              #F        *i+i/i#   r6        2
(14)            02              #T        *i+i/i#   S8
(15)            028             #T*       i+i/i#    S4
(16)            0284            #T*i      +i/i#     r8        13
(17)            028(13)         #T*F      +i/i#     r4        2
(18)            02              #T        +i/i#     r3        1
(19)            01              #E        +i/i#     S6
(20)            016             #E+       i/i#      S4
(21)            0164            #E+i      /i#       r8        3
(22)            0163            #E+F      /i#       r6        11
(23)            016(11)         #E+T      /i#       S9
(24)            016(11)9        #E+T/     i#        S4
(25)            016(11)94       #E+T/i    #         r8        14
(26)            016(11)9(114    #E+T/F    #         r5        11
(27)            016(11)9((11)   #E+T      #         r1        1
(28)            016(11)1        #E        #         acc
```

`ACTION`列最后显示`acc`，说明该表达式正确，成功被接收。

3. 对于算术表达式`i++i*i`，文法分析输出如下：

```
============================Parsing syntax using LR(0)============================
Step      Status Stack    Symbol Stack    Input    ACTION    GOTO
(1)             0               #         i++i*i#   S4
(2)             04              #i        ++i*i#    r8        3
(3)             03              #F        ++i*i#    r6        2
(4)             02              #T        ++i*i#    r3        1
(5)             01              #E        ++i*i#    S6
(6)             016             #E+       +i*i#     Row'6' Col'+' is empty!
```

当遇到`++`时，在LR0分析表中无法查找到该状态应该如何转换，提示相关报错信息，分析终止。不同于LL1分析器的是，本次实验中并没有在LR0分析器中加入错误处理的功能，因此LR0分析器遇到同样错误时错误时分析结束。

4. 对于算术表达式`i-i/i+i^i`，文法分析输出如下：

```
============================Parsing syntax using LR(0)============================
Step      Status Stack    Symbol Stack    Input    ACTION    GOTO
(1)             0               #         i-i*i^i#  S4
(2)             04              #i        -i*i^i#   r8        3
(3)             03              #F        -i*i^i#   r6        2
(4)             02              #T        -i*i^i#   r3        1
(5)             01              #E        -i*i^i#   S7
(6)             017             #E-       i*i^i#    S4
(7)             0174            #E-i      *i^i#     r8        3
(8)             0173            #E-F      *i^i#     r6        12
(9)             017(12)         #E-T      *i^i#     S8
(10)            017(12)8        #E-T*     i^i#      S4
(11)            017(12)84       #E-T*i    ^i#                 -48
(12)            017(12)84       #E-T*iE   i#
```

可以看到，对于`^`，它并不在我们所设计的文法中。因此分析出错，最后也没有`acc`，表示接收失败。

可见，实验中所设计的LR0文法分析器可以正确进行文法分析和错误判断。

7 实验感想

这次实验用LL1和LR两种方法实现了语法分析的功能，通过实验实现，对它们的基本原理有了更清楚的理解和掌握，对于分析表的构造以及分析过程的实现也学习得更加深入。

对比两种文法，它们都是无二义性的，LL(1)要求生成的预测分析表的每一个项目至多只能有一个生成式，即对于读头下的每一个字符，都可以明确地选择哪个产生式来推导；LR文法要求每一步都有明确的动作，移进和归约都是可确定的，同样没有二义性。LL(1)分析法是自上而下的分析法，即从开始符号出发，根据产生式规则推导给定的句子，用的是推导；LR分析法是自下而上的分析法，即从给定的句子规约到文法的开始符号，用的是规约。

本次实验还可以进一步提升的地方是可以与实验一的词法分析器结合起来，即先通过词法分析对输入的字符串进行识别，区别出一个个单词，并标以种别码，然后将其作为LL1分析器、LR分析器的输入，由于是将输入的单词和运算符作为字符串处理，这样就不仅仅能识别`i`作为表达式输入，而且可以识别数字（包括整数和小数）和变量等多种符号在内的运算，使功能相对而言更全面，这也是之后进一步实验要优化的地方。