

词法分析实验报告

池纪君 2019213688 2019211301班

北京邮电大学 计算机学院

实验题目

实验内容

语法分析程序的设计与实现

实验要求

设计要求

编写语法分析程序，实现对算术表达式的语法分析。要求所分析算术表达式由如下文法产生。

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid num$$

输出要求

在对输入的算术表达式进行分析过程中，依次输出所采用产生。

实现方法要求

1. 编写递归调用程序实现自顶向下的分析。
2. 编写 LL(1) 语法分析程序。
3. 编写语法分析程序实现自底向上的分析。
4. 利用 YACC 自动生成语法分析程序，调用 LEX 自动生成的词法分析程序。

程序设计说明

实验环境

1. 本实验在 Windows WSL2 上进行开发，其中 C 语言环境为 gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0 和 cmake version 3.16.3。
2. 编译&运行命令：

1. 方法1-3:

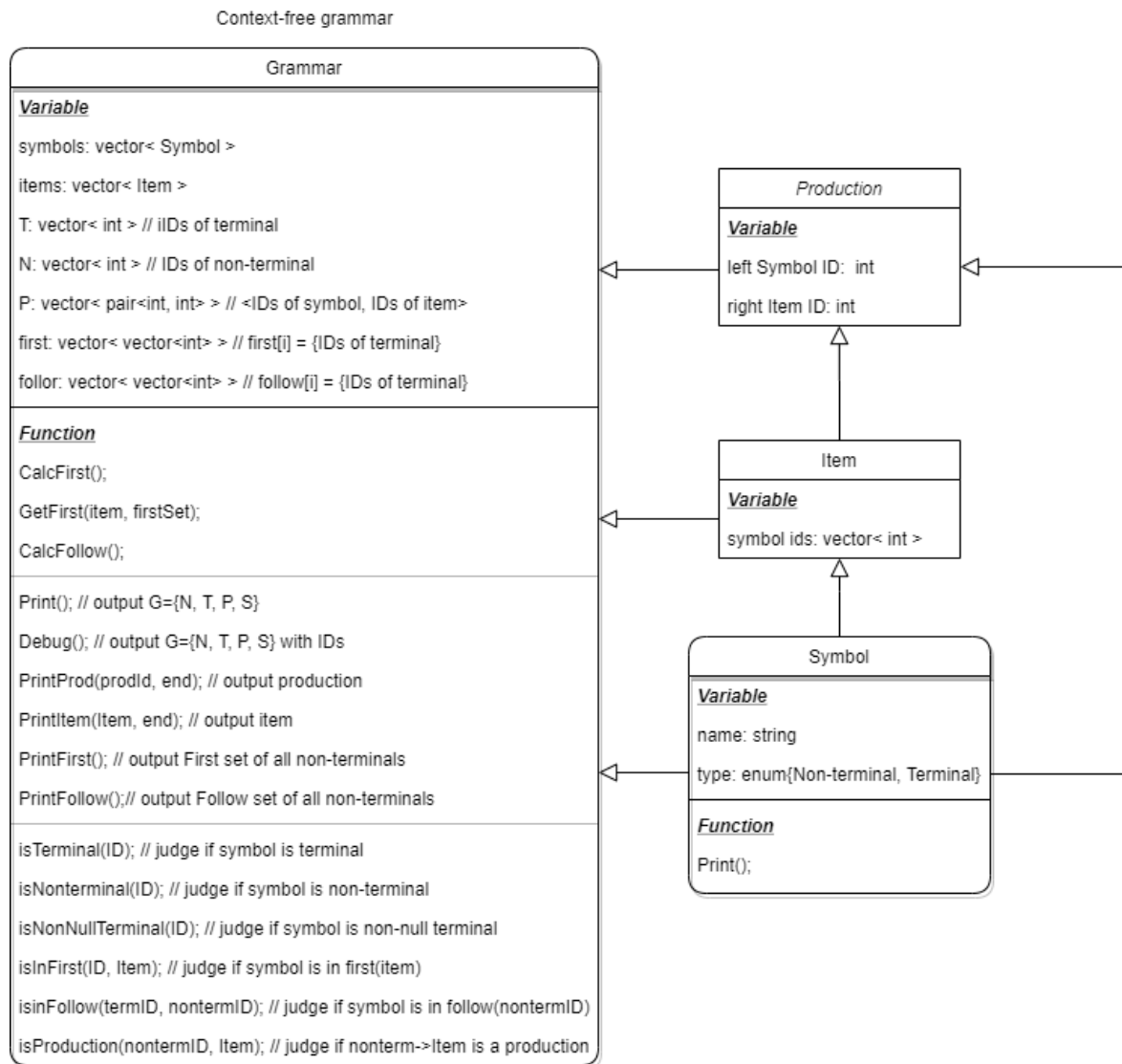
1. `cd parse`
2. `mkdir build && cd build`
3. `cmake ..`
4. `make && cd ..`
5. `./build/LL1test`、`./build/LRtest`

2. 方法4:

1. `cd yacc`
 2. `make`
 3. `./calc < test.p`
3. 额外的Linux可执行文件: `parse/_exe/*`、`yacc/calc`

文法的存储和相关算法的设计与实现

文法类的设计



消除左递归

1. 问题: 消除左递归的目的是为了防止在递归下降分析和LL分析的过程中产生无穷递归的死循环问题。

2. 算法：对文法 $G = (V_T, V_N, S, \varphi)$ ，可以用如下算法消除左递归：

Algorithm 1: EliminateLeftRecursion(Grammar \mathcal{G})	
Data: \mathcal{G} as input grammar	
Result: \mathcal{G}' as output grammar without left recursion	
1	Assign non-terminal characters: A_1, A_2, \dots, A_n ;
2	for $i \leftarrow 1$ to $ V_T $ do
3	// eliminate direct left recursion
4	$\forall A_i \rightarrow A_i \alpha_1 \dots A_i \alpha_n \beta_1 \dots \beta_p \in \varphi$, turn it to
	$A_i \rightarrow \beta_1 \dots \beta_p A'_i, A'_i \rightarrow \alpha_1 \dots \alpha_n \alpha_1 A'_i \dots \alpha_n A'_i$;
5	$i++$;
6	for $j \leftarrow 1$ to $i - 1$ do
7	for $A_i \rightarrow A_j \alpha$ in φ do
8	if $A_j \rightarrow \beta_1 \dots \beta_n$ in φ then
9	replace $A_i \rightarrow A_j \alpha$ by $A_i \rightarrow \beta_1 \alpha \beta_2 \alpha \dots \beta_n \alpha$;
10	end
11	end
12	end
13	$\mathcal{G}' = \text{EliminateUselessSymbol}(\text{Grammar } \mathcal{G})$
14	end
15	return \mathcal{G}' ;

提取左公因子

1. 问题：提取左部公因子的目的是为了使得LL(1)的每个非终结符（包括新产生的非终结符）的所有候选式首部符号两两不相交。
2. 算法：

- 如有产生式 $A \rightarrow \alpha \beta_1 | \alpha \beta_2$ 提取左公因子 α ，则原产生式变为：
$$\begin{array}{l} A \rightarrow \alpha A' \\ A' \rightarrow \beta_1 | \beta_2 \end{array}$$
- 若有产生式 $A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \dots | \alpha \beta_n | \gamma$ 可用如下的产生式代替：
$$\begin{array}{l} A \rightarrow \alpha A' | \gamma \\ A' \rightarrow \beta_1 | \beta_2 | \dots | \beta_n \end{array}$$

计算 FIRST 集合

定义

$\text{FIRST}(\alpha_i) = \{a \mid \alpha_i \xRightarrow{*} a\beta, a \in V_T, \alpha_i, \beta \in (V_T \cup V_N)^*\}$ ，表示可由 α_i 推导出的所有开头终结符号的集合。特别地，如果 $\alpha_i \xRightarrow{*} \varepsilon$ ，则规定 $\varepsilon \in \text{FIRST}(\alpha_i)$ 。

算法:

Algorithm 2: CalcFirst(Grammar \mathcal{G})

Data: \mathcal{G} as context-free grammar

Result: *First* as all terminal IDs non-terminal can firstly deduce

```
1 first  $\leftarrow \square$ ;
2 while True do
3   newFirst = first;
4   for Production  $p_i$  in  $\mathcal{G}$  do
5     if  $p_i = X \rightarrow a\ldots, a \in V_T$  then
6       | add  $\{a\}$  to first(X);
7     end
8     if  $p_i = X \rightarrow \epsilon$  then
9       | add  $\{\epsilon\}$  to first(X);
10    end
11    if  $p_i = X \rightarrow Y_1Y_2\ldots Y_k$  then
12      for  $i \leftarrow 1$  to  $k$  do
13        | if  $\epsilon \in \{\bigcap_{j=1}^{i-1} first(Y_j)\}$  then
14          | | add  $\{first(Y_i) - \{\epsilon\}\}$  to first(X);
15        | end
16      end
17      if  $\epsilon \in \{\bigcap_{j=1}^{i-1} first(Y_j)\}$  then
18        | add  $\{\epsilon\}$  to first(X);
19      end
20    end
21  end
22  if newFirst = first then
23    | break;
24  end
25 end
26 return first;
```

计算结果

计算样例文法的First集，调用 `Grammar.PrintFirst()`

```
1 First:
2     E: ( num
3     T: ( num
4     F: ( num
```

计算Follow集合

1. 定义: $FOLLOW(A) = \{a \mid S \xRightarrow{*} \cdots Aa\cdots, a \in V_T\}$, 表示该文法的所有句型中紧跟在 A 后的出现的终结符号或 $\$$ 组成的集合。特别地, 若 $S \xRightarrow{*} \cdots A$, 则规定 $\$ \in FOLLOW(A)$, $\$$ 为输入字符串的右结尾标志符。

2. 算法:

Algorithm 3: CalcFollow(Grammar \mathcal{G})

Data: \mathcal{G} as context-free grammar
Result: *Follow* as all terminal IDs non-terminal can followed by

```
1 follow(X)  $\leftarrow \emptyset$ ;  
2 add {$} to follow( $\mathcal{G}.S$ );  
3 while True do  
4     newFollow = follow;  
5     for Production  $p_i$  in  $\mathcal{G}$  do  
6         if  $p_i = A \rightarrow \alpha B \beta, \alpha, \beta \in V_T$  then  
7             | add first( $\beta$ ) - { $\epsilon$ } to first(B);  
8         end  
9         if  $p_i = A \rightarrow \alpha B$  or ( $p_i = A \rightarrow \alpha B \beta$ ) and  $\beta \xRightarrow{*} \epsilon$  then  
10            | add follow(A) to follow(B);  
11        end  
12    end  
13    if newFollow = follow then  
14        | break;  
15    end  
16 end  
17 return follow;
```

计算结果

计算样例文法的Follow, 调用 Grammar.PrintFollow()

```
1 Follow:  
2     E: $ + - )  
3     T: $ + - * / )  
4     F: $ + - * / )
```

构造拓广文法

定义

对任何文法 $G = (V_T, V_N, S, \varphi)$, 都有等价的文法:
 $G' = (V_T, V_N \cup \{S'\}, S', \varphi \cup \{S' \rightarrow S\})$ 称 G' 为 G 的拓广文法。拓广文法 G' 的接受项目是唯一的 (即 $S' \rightarrow S^*$)。

计算结果

计算样例文法的拓广文法, 调用 LRParser.GetExtG()

```
1 Grammar outline:  
2 Symbols:  
3     id: 0 1, $ Terminal  
4     id: 1 2,  $\epsilon$  Terminal  
5     id: 2 1, + Terminal  
6     id: 3 1, - Terminal  
7     id: 4 1, * Terminal  
8     id: 5 1, / Terminal  
9     id: 6 1, ( Terminal
```

```

10         id: 7 1, ) Terminal
11         id: 8 3, num Terminal
12         id: 9 1, E Nonterminal
13         id: 10 1, T Nonterminal
14         id: 11 1, F Nonterminal
15         id: 12 2, S' Nonterminal START
16     Productions:
17         id: 0, E ->E+T
18         id: 1, E ->E-T
19         id: 2, E ->T
20         id: 3, T ->T*F
21         id: 4, T ->T/F
22         id: 5, T ->F
23         id: 6, F ->(E)
24         id: 7, F ->num
25         id: 8, S' ->E

```

递归调用分析程序的设计与实现

递归下降分析

定义及工作过程

从文法的开始符号出发，进行推导，试图推出要分析的输入串的过程。对给定的输入符号串，从对应于文法开始符号的根结点出发，自顶向下地为输入串建立一棵分析树。

□ 试探过程，反复使用不同产生式谋求匹配输入符号串。

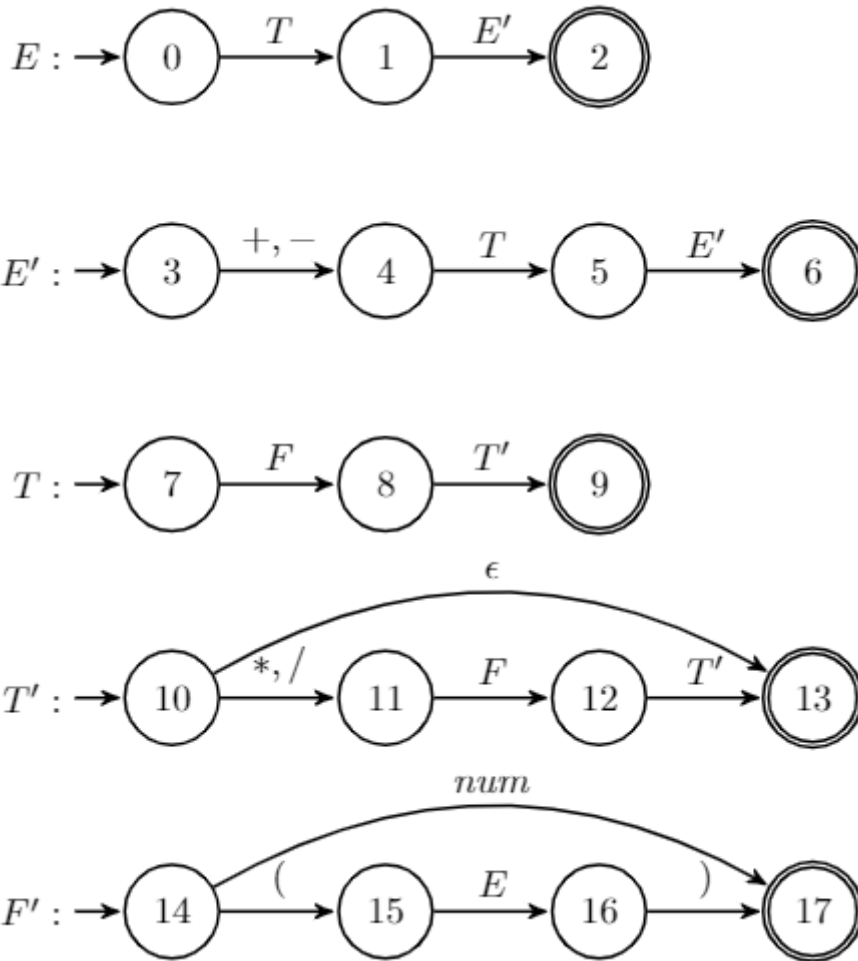
文法要求

1. 不含左递归
2. $\forall A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n, First(\alpha_i) \cap First(\alpha_j) = \Phi (i \neq j)$

预测分析程序构造

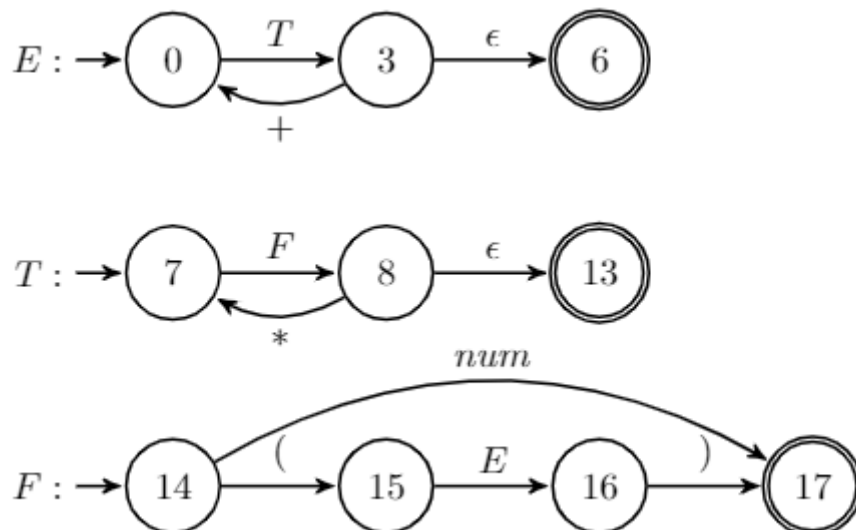
1. 构造转换图

在一个非终结符号A上的转移意味着对相应A的过程的调用。在一个终结符号a上的转移，意味着下一个输入符号若为a，则应做此转移。本次实验文法的各非中介符号构造转换图如下：



2. 转换图化简

将转换图进行带入化简，简化图如下：



3. 分析程序的实现

将递归分析程序存为RecurParser类，并将上图转换图封装成函数 `ProcE()`，`ProcF()`，`ProcT()`。分析程序收到词法分析结果串后，调用 `Analysis()` 开始递归分析。若成功，则会输出 `Accept`，否则输出 `Error`，并告知错误位置。

```
1 class RecurParser {
```

```

2     typedef vector<int> Item;
3     public:
4         Grammar gram;
5         RecurParser(Grammar gm) : gram(gm) {}
6         RecurParser() {}
7         bool Analysis(Item inp);
8
9     private:
10        Item input;
11        Symbol nowS;
12        int pos;
13        inline void Error() {
14            printf("error at position %d : ", pos);
15            nowS.print("\n");
16            exit(0);
17        }
18        void ProcE();
19        void ProcF();
20        void ProcT();
21        inline void fp() { nowS = gram.symbols[input[pos++]]; }
22    };
23

```

LL(1) 语法分析程序的设计与实现

LL(1)文法

定义

- 如果一个文法的预测分析表M不含多重定义的表项，则称该文法为LL(1)文法。
- LL(1)的含义：
第一个L表示从左至右扫描输入符号串。第二个L表示生成输入串的一个最左推导。1表示在决定分析程序的每步动作时，向前看一个符号。

判断方法

1. 一个文法是LL(1)文法，当且仅当它的每一个产生式 $A \rightarrow \alpha$ ，满足： $\text{First}(\alpha) \cap \beta = \phi$ ，并且若 β 推导出 ϵ ，则 $\text{Follow}(A) = \phi$ 。
2. 如果利用算法4.2构造出的分析表中不含多重定义的表项，则文法是LL(1)文法。

预测分析表的构造算法

1. 错误处理：
 - 对于 $A \in \mathbf{V}_N$ ， $b \in \text{FOLLOW}(A)$ ，若 $M[A,b]$ 为空，则加入“synch”。
 - 在使用带有同步化信息的分析表时，当前读入a，状态为A。若 $M[A, a]$ 为空，则跳过a；若 $M[A, a]$ 为synch，则弹出A。
2. 算法

Algorithm 5: GetLL1Table(Grammar \mathcal{G})

Data: \mathcal{G} as input grammar
Result: M as analysis table

```
1 for Production  $A \rightarrow \alpha$  in  $\mathcal{G}.Production$  do
2   for Terminal  $term \in first(\alpha)$  do
3     | add  $A \rightarrow \alpha$  to  $M[A,a]$ ;
4   end
5   if  $\epsilon \in first(\alpha)$  then
6     | for Terminal  $b \in follow(A)$  do
7       | | add  $A \rightarrow \alpha$  to  $M[A,b]$ ;
8     | end
9   end
10 end
11 for Non-terminal  $A \in \mathcal{G}.NonTerminal$  do
12   for Terminal  $b \in \mathcal{G}.Terminal$  do
13     | if  $M[A,b]$  is null then
14       | | if  $b \in follow(A)$  then
15       | | |  $M[A,b] = SYNC$ ;
16       | | else
17       | | |  $M[A,b] = ERROR$ 
18       | | end
19     | end
20   end
21 end
22 return  $M$ ;
```

LL(1)非递归预测分析

预测分析分析程序模型及工作流程

1. 程序组成部分:

1. 输入缓冲区: 存放被分析的输入符号串, 串后随右尾标志符。
2. 符号栈: 存放一系列文法符号, 存于栈底。分析开始时, 先将 $\$$ 入栈, 以标识栈底, 然后再将文法的开始符号入栈。
3. 分析表: 二维数组 $M[A,a]$, $A \in V_N$, $a \in V_T \cup \{\$\}$ 。根据给定的 A 和 a , 在分析表 M 中找到将被调用的产生式。
4. 输出流: 分析过程中不断产生的产生式序列。

2. 工作流程:

3. 根据栈顶符号 X 和当前输入符号 a , 分析动作有4种可能:

- (1) $X = a = \$$, 宣告分析成功, 停止分析;
- (2) $X = a \neq \$$, 从栈顶弹出 X , 输入指针前移一个位置;
- (3) $X \in V_T$, 但 $X \neq a$, 报告发现错误, 调用错误处理程序, 报告错误及进行错误恢复;
- (4) 若 $X \in V_N$, 访问分析表 $M[X,a]$

1. $M[X,a] = XY_1Y_2 \dots Y_n$, 先将 X 从栈顶弹出, 然后把产生式的右部符号串按反序 推入栈中 (即按 $\$Y_n, \dots, Y_2, Y_1$ 的顺序) ;
2. $M[X,a] = X \rightarrow \epsilon$ 从栈顶弹出 X ;
3. $M[X,a] = ERROR$ 调用出错处理程序, (恢复: $M[X,a] = sync$ 弹出栈顶 A , 否则跳过a)

算法

Algorithm 6: LL1Analysis(M, ω)

Data: M as analysis table, ω as input symbol string

Result: output as the leftmost derivation of ω

```
1  $p \leftarrow \omega.\text{first};$ 
2  $stk \leftarrow []; X \leftarrow \omega[p]$ 
3 while  $X \neq \$$  do
4    $X \leftarrow \text{stk.front}();$ 
5    $a \leftarrow \omega[p];$ 
6   if  $X$  is terminal or  $X$  is  $\$$  then
7     if  $X == a$  then
8        $\text{stk.pop}();$ 
9        $p++;$ 
10    else
11       $\text{error}();$ 
12    end
13  else
14    if  $M[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$  then
15       $\text{stk.pop}(); \text{stk.push}(Y_k, Y_{k-1}, \dots, Y_2, Y_1);$ 
16    else
17       $\text{error}();$ 
18    end
19  end
20 end
```

类设计

```
1 class LL1Parser : public Parser {
2     typedef vector<int> Item;
3
4     private:
5         // id of Nonterminal, Terminal and production
6         map<pair<int, int>, int> LL1Table;
7
8         void PrintStack(stack<int>& stk, const char* end);
9         void PrintVec(vector<int>& vec, int start, const char* end);
10
11     public:
12         static const int ERROR = -2;
13         static const int SYNC = -1;
14         LL1Parser(Grammar gm) : Parser(gm){};
15         bool GetLL1Table(); // return false means conflict
16         void PrintTable();
17
18         // conduct a LL(1) analysis
19         bool LL1Analysis(Item input);
20 };
21
```

自底向上分析的语法分析程序的设计与实现

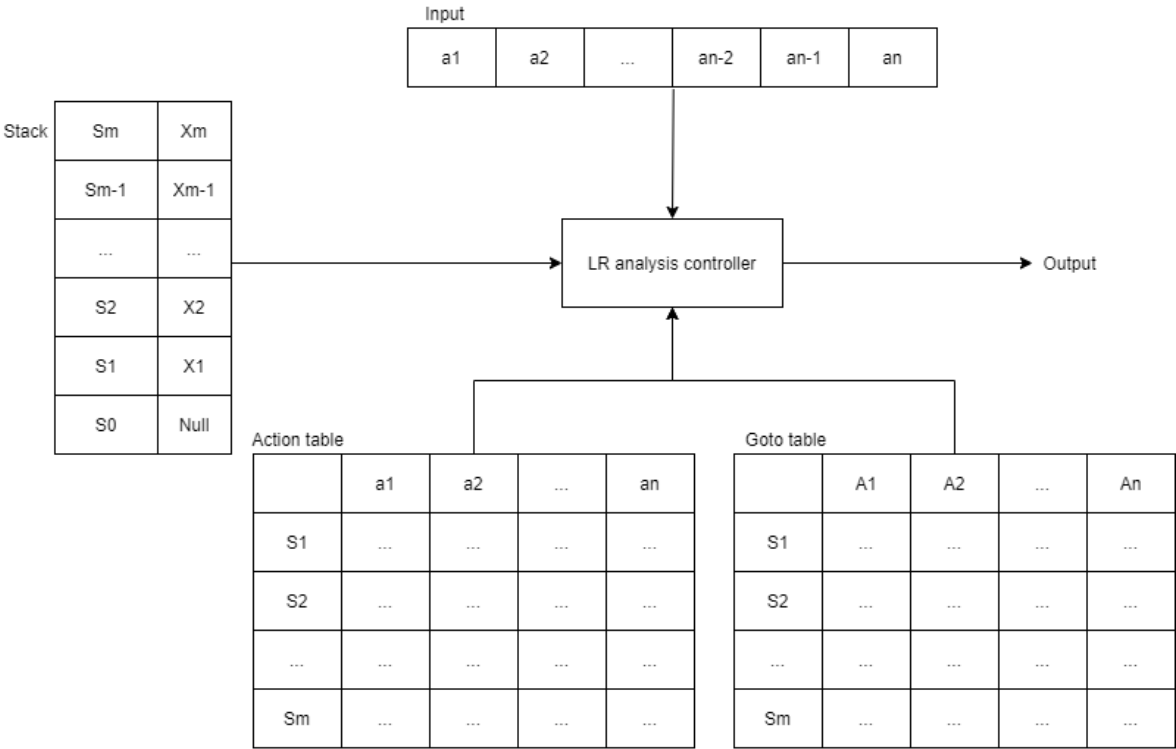
LR(1)分析技术

定义

- 1. LR(k)的含义：
 - L 表示自左至右扫描输入符号串
 - R 表示为输入符号串构造一个最右推导的逆过程
 - k 表示为作出分析决定而向前看的输入符号的个数
- 2. LR分析方法的基本思想
 - 记住已经移进和归约出的整个符号串--历史信息；
 - 根据所用的产生式推测未来可能遇到的输入符号--预测信息；

根据“历史信息”和“预测信息”，以及“现实”的输入符号，确定栈顶的符号串是否构成相对于某一产生式的句柄。

工作模型



SLR(1)分析表的构造

LR(0)有效项目

- 1. LR(0)项目
 - 1. 定义：右部某个位置上标有圆点的产生式称为文法G的一个LR(0)项目，例如： $A \rightarrow \cdot XYZ$, $A \rightarrow X \cdot YZ$, $A \rightarrow XY \cdot Z$, $A \rightarrow XYZ \cdot$.
- 2. 类别：
 - 1. 归约项目：圆点在产生式最右端的LR(0)项目
 - 2. 接受项目：对文法开始符号的归约项目
 - 3. 待约项目：圆点后第一个符号为非终结符号的LR(0)项目
 - 4. 移进项目：圆点后第一个符号为终结符号的LR(0)项目

2. LR(0)有效项目

1. 定义：项目 $A \rightarrow \beta_1 \cdot \beta_2$ 对活前缀 $\gamma = \alpha\beta$ 是有效的，如果存在一个规范推导：
 $S \xRightarrow{*} \alpha A \omega \Rightarrow \alpha \beta_1 \beta_2 \omega$ 。LR(0)项目 $S' \rightarrow \cdot S$ 是活前缀 ϵ 的有效项目。
2. LR(0)有效项目集：文法G的某个活前缀的所有LR(0)有效项目组成的集合。
3. LR(0)项目集规范族：文法G的所有LR(0)有效项目集组成的集合。

SLR(1)分析方法

1. 构造 SLR(1)分析表的基本思想是：首先为给定文法构造一个识别它的所有活前缀的确定的有限自动机,然后根据此有限自动机构造该文法的分析表。
2. SLR(1)的特点：可以通过考察follow集合解决冲突。
 1. 冲突： $I = \{X \rightarrow \alpha \cdot b\beta, A \rightarrow \alpha \cdot, B \rightarrow \beta \cdot\}$ 存在移进-归约冲突和归约-归约冲突。
 2. 解决：对于当前读入a, 查看FOLLOW(A)和FOLLOW(B)。(要求： $FOLLOW(A) \cap FOLLOW(B) = \Phi$, $b \notin FOLLOW(A)$ 并且 $b \notin FOLLOW(B)$)
 1. 当 $a = b$ 时, 把b移进栈里;
 2. 当 $a \in FOLLOW(A)$ 时, 用产生式 $A \rightarrow \alpha$ 进行归约;
 3. 当 $a \in FOLLOW(B)$ 时, 用产生式 $B \rightarrow \alpha$ 进行归约。
3. 构造SLR(1)分析表

1. 闭包closure(I):

设I是文法G的一个LR(0)项目集合, closure(I)是从I出发, 用下面的方法构造的项目集:

1. I中的每一个项目都属于closure(I);
 2. 若项目 $A \rightarrow \cdot B\beta$ 属于closure(I), 且G有产生式 $B \rightarrow \eta$, 若 $B \rightarrow \cdot \eta$ 不属于closure(I), 则将 $B \rightarrow \cdot \eta$ 加入 closure(I);
 3. 重复规则 ii, 直到closure(I)不再增大为止。
2. 转移函数go:
若I是文法G的一个LR(0)项目集, X是一个文法符号, 定义 $go(I, X) = closure(J)$, 其中 $J = \{A \rightarrow \alpha X \cdot \beta \mid \text{当 } A \rightarrow \alpha \cdot X\beta \text{ 属于 } I \text{ 时}\}$ 。go(I, X)为转移函数, 项目 $A \rightarrow \alpha X \cdot \beta$ 称为 $A \rightarrow \cdot X\beta$ 的后续。

算法

1. 文法的LR(0)项目集规范族的构造算法

Algorithm 7: GetLR0Fmly(\mathcal{G}')

Data: \mathcal{G}' as extended grammar
Result: fmly as LR(0) item sets

```
1 fmly = {Closuer({ $S' \rightarrow S$ })}
2 repeat
3   newFmly = fmly; for Set I in fmly do
4     for Symbol X in  $\mathcal{G}'$  do
5       newSet = go(I,X); if newSet  $\neq \Phi$  and newSet  $\notin$  fmly then
6         add newSet to fmly;
7       end
8     end
9   end
10 until newFmly != fmly;
11 return fmly;
```

2. 求解闭包Closure算法

Algorithm 8: Closure(\mathcal{I})

Data: \mathcal{I} as item set
Result: \mathcal{S} as closure(\mathcal{I})

```
1  $\mathcal{S} = \mathcal{I}$ ; repeat
2   newSet =  $\mathcal{S}$ ; for  $A \rightarrow \alpha \cdot B\beta \in \text{newSet}$  and  $B \rightarrow \eta \in \text{Grammar } G$ 
3     do
4       if  $B \rightarrow \cdot \eta \notin \mathcal{S}$  then
5         | add  $B \rightarrow \cdot \eta$  to  $\mathcal{S}$ ;
6       end
7     end
8 until newSet !=  $\mathcal{S}$ ;
return set;
```

3. SLR(1)分析表的构造算法

Algorithm 9: GetSLRTable(\mathcal{G}')

Data: \mathcal{G}' as extended Grammar
Result: action, goto as analysis table of \mathcal{G}'

```
1 fmly = GetLR0Fmly( $\mathcal{G}'$ ); // Algorithm 7
2 for Set  $I_i$  in fmly do
3   if  $\mathbf{A} \rightarrow \alpha \cdot \mathbf{a}\beta \in I_i$  and  $go(I_i, a) = I_j$  then
4     | action[i,a] =  $S_j$ ;
5   end
6   if  $\mathbf{A} \rightarrow \alpha \cdot \in I_i$  then
7     | action[i,a] =  $R_{A \rightarrow \alpha}, \forall a \in Follow(A)$ ;
8   end
9   if  $\mathbf{S}' \rightarrow \mathbf{S} \cdot \in I_i$  then
10    | action[i,$] = ACC;
11  end
12  if  $go(I_i, A) = I_j, A \in \mathcal{G}'.T$  then
13    | goto[i,A] = j;
14  end
15 end
16 return action, goto;
```

LR分析控制程序

算法设计

Algorithm 10: SLRAnalysis(Analysis table, ω)

Data: action, goto as analysis table, ω as input symbols

Result: output a bottom-up analysis

```
1  $pos \leftarrow \omega[0]$ ;
2  $stk \leftarrow []$ ;
3 push  $\langle null, S' \rangle$  to  $stk$ ;
4 repeat
5    $S \leftarrow stk.top().second$ ;
6    $a = \omega[pos]$ ;
7   if  $action[S, a] = SHIFT\ S'$  then
8     push  $\langle a, S' \rangle$  to  $stk$ ;
9      $pos++$ ;
10  else
11    if  $action[S, a] = REDUCE\ A \rightarrow \beta$  then
12       $stk$  pop  $|\beta|$  times;
13       $S' \leftarrow stk.top().second$ ;
14      push  $\langle A, goto[S', A] \rangle$  to  $stk$ ;
15      output  $A \rightarrow \beta$ ;
16    else
17      if  $action[S, a] = ACCEPT$  then
18        return;
19      else
20        ERROR();
21      end
22    end
23  end
24  return;
25 until True;
```

类设计

```
1 class LRParser : public Parser {
2   typedef vector<int> Item;
3   typedef pair<int, Item> LR0Item; // LR(0) item: id of nonterminal, Item
4   typedef vector<LR0Item> LR0Set; // LR(0) item set
5   typedef vector<LR0Set> LR0Fmly; // LR(0) item set specification family
6
7   const int DOT_ID = -1;
8
9   private:
10    bool isExt;
11
12   public:
13    enum ActionType { SHIFT, REDUCE, ACCEPT, ERROR } actionType;
14    Grammar extGram;
15
16   private:
17    LR0Fmly lR0Fmly;
18
19    // <state id, nonterminal id> -> next state id
20    map<pair<int, int>, int> gotoTable;
```

```

21
22 // <state id, terminal id> -> <action type, action code>
23 map<pair<int, int>, pair<ActionType, int>> actionTable;
24
25 // get extended grammar
26 void GetExtG();
27
28 // closure{nowItem} => nowSet
29 void GetClosure(LR0Item item, LR0Set& set);
30
31 // go(set, x) => newSet
32 void GetGo(LR0Set nowSet, int x, LR0Set& newSet);
33
34 void PrintLR0Item(const LR0Item& lR0Item, const char* end);
35 void PrintLR0Set(const LR0Set& lR0Set, const char* end);
36
37 // print LR(0) set family and DFA
38 void PrintLR0Fmly();
39
40 void PrintStk(vector<pair<int, int>>& stk, const char* c);
41
42 public:
43 LRParser(Grammar gm) : Parser(gm) {
44     isExt = false;
45     GetExtG();
46 };
47 ~LRParser(){};
48
49 // Get SLR(1) analysis table
50 void GetSLRTable();
51
52 void PrintLRTable();
53
54 // conduct a SLR(1) analysis
55 bool SLRAnalysis(Item input);
56 };
57

```

YACC自动生成语法分析程序

YACC简介

- yacc(Yet Another Compiler Compiler), 是Unix/Linux上一个用来生成编译器的编译器 (编译器代码生成器) 。
使用巴克斯范式(BNF)定义语法, 能处理上下文无关文法(context-free)。出现在每个产生式左边的符号是非终端符号, 出现在产生式右边的符号有非终端符号和终端符号, 但终端符号只出现在右端。
- yacc是开发编译器的一个有用的工具, 采用LR (1) (实际上是LALR(1)) 语法分析方法。这种方法具有分析速度快, 能准确, 即使地指出出错的位置, 它的主要缺点是一个使用语言文法的分析器的构造工作量相当大, k愈大构造愈复杂, 实现比较困难。

Yacc工作流程与和词法分析程序Lex间的通信

- Lex和Yacc的工作关系可以看作一个生产者-消费者模型。Lex对文件字符串进行解析，将解析到的符号通过 `yylex()` 函数传给Yacc程序，再由Yacc程序依次对收到的符号进行LR语法分析。
- 一个由 Yacc 生成的解析器调用 `yylex()` 函数来获得标记，只有在文件结束或者出现错误标记时才会终止。`yylex()` 可以由 Lex 来生成或完全由自己来编写。对于由 Lex 生成的 lexer 来说，要和 Yacc 结合使用，每当 Lex 中匹配一个模式时，都必须返回一个标记。因此 Lex 中匹配模式时的动作一般格式为：`{pattern} { /* do something */ return TOKEN_NAME; }`，Yacc获得返回的标记，进行下一步的LR语法归约。当 Yacc 编译一个带有 `-d` 标记的 `.y` 文件时，会生成一个 `y.tab.h` 的头文件，它对每个终结符标记进行 `#define` 定义。Lex 和 Yacc 一起使用的话，Lex 文件 `.lex` 必须在C声明段中包括 `y.tab.h`。

Yacc的规则

- Yacc文法说明文件可以分为三个段：

```
1  [第一部分：定义段]
2  %%
3  第二部分：规则段
4  %%
5  [第三部分：辅助函数段]
```

- 定义段可以分为两部分：

- 第一部分以符号`%{`和`%}`包裹，里面为以C语法写的一些定义和声明：例如，文件包含，宏定义，全局变量定义，函数声明等。
 - 第二部分主要是对文法的终结符和非终结符做一些相关声明。这些声明主要有如下一些：`%token`, `%left`, `%right`, `%nonassoc`, `%union`, `%type`, `%start`。下面分别说明它们的用法。
- 规则段实际上定义了文法的非终结符及产生式集合，以及当归约整个产生式时应执行的操作。假如产生式为`expr : expr plus term | term`，则在规则段应该写成：

```
1  expr: expr PLUS term      {语义动作}
2      | term                {语义动作}
3      ;
```

- 辅助函数段用C语言语法来写，辅助函数一般指在规则段中用到或者在语法分析器的其他部分用到的函数。这一部分一般会被直接拷贝到yacc编译器产生的c源文件中。一般来说，除规则段用到的函数外，辅助函数段一般包括如下一些例程：`yylex()`, `yyerror()`, `main()`。

- Yacc文件的使用

- 生成`y.tab.c`。

`bison --yacc -d -v filename.y` 其中 `-d` 表示生成名为`filename.tab.h`的头文件，`-v` 表示生成`filename.output`文件，该文件说明了该语法分析器使用的识别活前缀的DFA。

- 与lex联合编译

`flex filename.l` 将lex源文件编译成.c的文件格式，`gcc -o example y.tab.c lex.yy.c` 将lex和yacc代码共同编译生成 `example` 的可执行程序。（注：在 `filemae.l` 中应包含 `y.tab.h` 的头文件）

- 使用 `makefile` 简化编译过程：详见 `makefile` 文件。

- 进行语法分析

`./example < test.p` 即可将样例文件进行语法分析。

Yacc程序实现

1. calc.l

- 对数字和符号进行词法解析
- 提供 `getNum()` 函数，为语法分析提供解析到的数的值

2. calc.y

- 对接受的符号进行语法归约
- 在不同产生式的语义动作中，进行表达式的值的计算，并按照归约顺序输出归约使用的产生式。
- 输出表达式的值
- 在 `yyerror()` 中提供错误处理

测试报告

测试程序说明

- lex: 测试样例文法读取，存储，输出和词法分析正确性。文法文件: `data/grammar.in`，分析文件 `data/expression.in`。
- Rectest: 测试递归下降分析程序正确性，输出分析结果。
- LL1test: 测试用LL(1)文法进行预测分析的正确性，输出预测分析表，分析过程和分析结果。
- LRtest: 测试拓广文法正确性，输出拓广文法。测试SLR(1)文法进行语法分析的正确性，输出识别活前缀的项目集规范族DFA，预测分析表action和goto，输出LR分析过程和分析结果。
- LL1synthesis: 综合词法程序lex和LL语法分析程序，输出对文件 `data/expression.in` 的分析过程。
- LRsynthesis: 综合词法程序lex和LR语法分析程序，输出对文件 `data/expression.in` 的分析过程。

测试结果与说明

1. 数据文件:

1. expression.in

```
1 | (3.090+878*24.345)*.013445
```

2. grammar.in

```
1 Terminal:
2   + - * / ( ) num
3
4 Nonterminal:
5   E T F
6
7 Production:
8   E -> E+T
9   E -> E-T
10  E -> T
11  T -> T*F
```

```

12      T -> T/F
13      T -> F
14      F -> (E)
15      F -> num
16
17  Start:
18      E

```

3. grammar3.in (为grammar.in消去左递归)

```

1  Terminal:
2      + - * / ( ) num
3
4  Nonterminal:
5      E T F E' T'
6
7  Production:
8      E -> TE'
9      E' -> +TE'
10     E' -> -TE'
11     E' -> ε
12     T -> FT'
13     T' -> *FT'
14     T' -> /FT'
15     T' -> ε
16     F -> (E)
17     F -> num
18
19  Start:
20     E

```

2. 测试结果:

1. lex

```

1  fschi@FCXiaoXin:/mnt/v/Code/exp_compile/2_yacc/parse$ ./build/lex
2
3  Grammar info:
4  Terminal character:
5      $ ε + - * / ( ) num
6  Nonterminal character:
7      E T F
8  Production:
9      E -> E+T
10     E -> E-T
11     E -> T
12     T -> T*F
13     T -> T/F
14     T -> F
15     F -> (E)
16     F -> num
17  Start character: E
18
19  Grammar outline:
20  Symbols:
21      id: 0 1, $ Terminal
22      id: 1 2, ε Terminal

```

```

23         id: 2 1, + Terminal
24         id: 3 1, - Terminal
25         id: 4 1, * Terminal
26         id: 5 1, / Terminal
27         id: 6 1, ( Terminal
28         id: 7 1, ) Terminal
29         id: 8 3, num Terminal
30         id: 9 1, E Nonterminal START
31         id: 10 1, T Nonterminal
32         id: 11 1, F Nonterminal
33     Productions:
34         id: 0, E ->E+T
35         id: 1, E ->E-T
36         id: 2, E ->T
37         id: 3, T ->T*F
38         id: 4, T ->T/F
39         id: 5, T ->F
40         id: 6, F ->(E)
41         id: 7, F ->num
42     Input file: data/expression.in
43     6 8 2 8 4 8 7 4 8

```

2. Rectest

```

1  fschi@FCXiaoXin:/mnt/v/Code/exp_compile/2_yacc/parse$
   ./build/Rectest
2
3  Grammar info:
4  Terminal character:
5      $ € + - * / ( ) num
6  Nonterminal character:
7      E T F
8  Production:
9      E -> E+T
10     E -> E-T
11     E -> T
12     T -> T*F
13     T -> T/F
14     T -> F
15     F -> (E)
16     F -> num
17  Start character: E
18
19  Grammar outline:
20  Symbols:
21     id: 0 1, $ Terminal
22     id: 1 2, € Terminal
23     id: 2 1, + Terminal
24     id: 3 1, - Terminal
25     id: 4 1, * Terminal
26     id: 5 1, / Terminal
27     id: 6 1, ( Terminal
28     id: 7 1, ) Terminal
29     id: 8 3, num Terminal
30     id: 9 1, E Nonterminal START
31     id: 10 1, T Nonterminal
32     id: 11 1, F Nonterminal

```

```

33 Productions:
34     id: 0, E ->E+T
35     id: 1, E ->E-T
36     id: 2, E ->T
37     id: 3, T ->T*F
38     id: 4, T ->T/F
39     id: 5, T ->F
40     id: 6, F ->(E)
41     id: 7, F ->num
42 Accept!

```

3. LL1test

LL(1)分析表

LL1 analysis table:									
	\$	ε	+	-	*	/	()	num
E	SYNC						E -> TE'	SYNC	E -> TE'
T	SYNC		SYNC	SYNC			T -> FT'	SYNC	T -> FT'
F	SYNC		SYNC	SYNC	SYNC	SYNC	F -> (E)	SYNC	F -> num
E'	E' -> ε		E' -> +TE'	E' -> -TE'				E' -> ε	
T'	T' -> ε		T' -> ε	T' -> ε	T' -> *FT'	T' -> /FT'		T' -> ε	

测试结果:

```

1  fschi@FCxiaoXin:/mnt/v/Code/exp_compile/2_yacc/parse$
   ./build/LL1test
2
3  LL1 analysis table:
4
   | $      ε      +      -      *      /
   (      )      num
5  -----
6  E      | SYNC
   E -> TE'  SYNC      E -> TE'
7  T      | SYNC
   T -> FT'  SYNC      T -> FT'
8  F      | SYNC
   SYNC      F -> (E)  SYNC      F -> num
9  E'     | E' -> ε
           E' -> ε
10 T'     | T' -> ε
   T' -> /FT'      T' -> ε      T' -> ε      T' -> *FT'
11
12
13 Start nonrecursive analysis.
14 Stack: E $      Input: ( num + num ) * num $      Output: E -> T E'
15 Stack: T E' $   Input: ( num + num ) * num $      Output: T -> F T'
16 Stack: F T' E' $   Input: ( num + num ) * num $      Output: F -
   > ( E )
17 Stack: ( E ) T' E' $   Input: ( num + num ) * num $      eliminate (
18 Stack: E ) T' E' $   Input: num + num ) * num $      Output: E -
   > T E'
19 Stack: T E' ) T' E' $   Input: num + num ) * num $      Output: T -
   > F T'
20 Stack: F T' E' ) T' E' $   Input: num + num ) * num $
   Output: F -> num
21 Stack: num T' E' ) T' E' $   Input: num + num ) * num $
   eliminate num
22 Stack: T' E' ) T' E' $   Input: + num ) * num $      Output: T' -> ε

```

```

23 Stack: E' ) T' E' $      Input: + num ) * num $      Output: E' -> + T
    E'
24 Stack: + T E' ) T' E' $      Input: + num ) * num $      eliminate +
25 Stack: T E' ) T' E' $      Input: num ) * num $      Output: T -> F T'
26 Stack: F T' E' ) T' E' $      Input: num ) * num $      Output: F -
    > num
27 Stack: num T' E' ) T' E' $      Input: num ) * num $      eliminate
    num
28 Stack: T' E' ) T' E' $      Input: ) * num $      Output: T' -> ε
29 Stack: E' ) T' E' $      Input: ) * num $      Output: E' -> ε
30 Stack: ) T' E' $      Input: ) * num $      eliminate )
31 Stack: T' E' $      Input: * num $      Output: T' -> * F T'
32 Stack: * F T' E' $      Input: * num $      eliminate *
33 Stack: F T' E' $      Input: num $      Output: F -> num
34 Stack: num T' E' $      Input: num $      eliminate num
35 Stack: T' E' $      Input: $      Output: T' -> ε
36 Stack: E' $      Input: $      Output: E' -> ε
37 Stack: $      Input: $      eliminate $
38 Analysis succeed.

```

4. LRtest

LR(0)项目集规范族:

```

LR0 DFA:
{
Set 0: { [S' -> .E], [E -> .E+T], [E -> .E-T], [E -> .T], [T -> .T*F], [T -> .T/F], [T -> .F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 1: { [F -> .(E)], [E -> .E+T], [E -> .E-T], [E -> .T], [T -> .T*F], [T -> .T/F], [T -> .F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 2: { [F -> .num] }
Set 3: { [E -> E+T], [E -> E-T], [S' -> E.] } | read '+' -> set 7, '-' -> set 8
Set 4: { [E -> T.], [T -> T.*F], [T -> T./F] } | read '*' -> set 9, '/' -> set 10
Set 5: { [T -> F.] }
Set 6: { [E -> E+T], [E -> E-T], [F -> .(E)] } | read '(' -> set 7, '-' -> set 8, ')' -> set 11
Set 7: { [E -> E+T], [T -> .T*F], [T -> .T/F], [T -> .F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 8: { [E -> E-T], [T -> .T*F], [T -> .T/F], [T -> .F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 9: { [T -> T*.F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 10: { [T -> T/F.F], [F -> .(E)], [F -> .num] } | read '(' -> set 1, 'num' -> set 2
Set 11: { [F -> .(E.)] }
Set 12: { [E -> E+T.], [T -> T.*F], [T -> T./F] } | read '*' -> set 9, '/' -> set 10
Set 13: { [E -> E-T.], [T -> T.*F], [T -> T./F] } | read '*' -> set 9, '/' -> set 10
Set 14: { [T -> T*F.] }
Set 15: { [T -> T/F.] }
}

```

SLR(1)分析表:

SLR(1) Table:													
State	action								goto				
	\$	+	-	*	/	()	num	E	T	F	E'	T'
0						S1		S2	3	4	5		
1						S1		S2	6	4	5		
2	R9	R9	R9	R9	R9		R9						
3	ACC												
4	R3	S7	S8				R3					9	
5	R7	R7	R7	S10	S11		R7						12
6							S13						
7						S1		S2		14	5		
8						S1		S2		15	5		
9	R0						R0						
10						S1		S2				16	
11						S1		S2				17	
12	R4	R4	R4				R4						
13	R8	R8	R8	R8	R8		R8						
14	R3	S7	S8				R3					18	
15	R3	S7	S8				R3					19	
16	R7	R7	R7	S10	S11		R7						20
17	R7	R7	R7	S10	S11		R7						21
18	R1						R1						
19	R2						R2						
20	R5	R5	R5				R5						
21	R6	R6	R6				R6						

测试结果:

```

1 fschi@FCxiaoxin:/mnt/v/Code/exp_compile/2_yacc/parse$
  ./build/LRtest
2
3 Grammar info:

```

```

4 Terminal character:
5     $ € + - * / ( ) num
6 Nonterminal character:
7     E T F E' T' S'
8 Production:
9     E -> TE'
10    E' -> +TE'
11    E' -> -TE'
12    E' -> €
13    T -> FT'
14    T' -> *FT'
15    T' -> /FT'
16    T' -> €
17    F -> (E)
18    F -> num
19    S' -> E
20 Start character: S'
21
22 Grammar outline:
23 Symbols:
24     id: 0 1, $ Terminal
25     id: 1 2, € Terminal
26     id: 2 1, + Terminal
27     id: 3 1, - Terminal
28     id: 4 1, * Terminal
29     id: 5 1, / Terminal
30     id: 6 1, ( Terminal
31     id: 7 1, ) Terminal
32     id: 8 3, num Terminal
33     id: 9 1, E Nonterminal
34     id: 10 1, T Nonterminal
35     id: 11 1, F Nonterminal
36     id: 12 2, E' Nonterminal
37     id: 13 2, T' Nonterminal
38     id: 14 2, S' Nonterminal START
39 Productions:
40     id: 0, E ->TE'
41     id: 1, E' ->+TE'
42     id: 2, E' ->-TE'
43     id: 3, E' ->€
44     id: 4, T ->FT'
45     id: 5, T' ->*FT'
46     id: 6, T' ->/FT'
47     id: 7, T' ->€
48     id: 8, F ->(E)
49     id: 9, F ->num
50     id: 10, S' ->E
51
52 Generating LR(0) item sets.
53 LR(0) item sets generated.
54
55 LR0 DFA:
56 {
57 Set 0: { [S' -> .E], [E -> .TE'], [T -> .FT'], [F -> .(E)], [F ->
    .num] } | read '(' -> set 1, 'num' -> set 2
58 Set 1: { [F -> (.E)], [E -> .TE'], [T -> .FT'], [F -> .(E)], [F ->
    .num] } | read '(' -> set 1, 'num' -> set 2
59 Set 2: { [F -> num.] }

```

```

60 Set 3: { [S' -> E.] }
61 Set 4: { [E -> T.E'], [E' -> .+TE'], [E' -> .-TE'], [E' -> .] } |
    read '+' -> set 7, '-' -> set 8
62 Set 5: { [T -> F.T'], [T' -> .*FT'], [T' -> ./FT'], [T' -> .] } |
    read '*' -> set 10, '/' -> set 11
63 Set 6: { [F -> (E.)] } | read ')' -> set 13
64 Set 7: { [E' -> +.TE'], [T -> .FT'], [F -> .(E)], [F -> .num] } |
    read '(' -> set 1, 'num' -> set 2
65 Set 8: { [E' -> -.TE'], [T -> .FT'], [F -> .(E)], [F -> .num] } |
    read '(' -> set 1, 'num' -> set 2
66 Set 9: { [E -> TE'.] }
67 Set 10: { [T' -> *.FT'], [F -> .(E)], [F -> .num] } | read '(' -
    > set 1, 'num' -> set 2
68 Set 11: { [T' -> /.FT'], [F -> .(E)], [F -> .num] } | read '(' -
    > set 1, 'num' -> set 2
69 Set 12: { [T -> FT'.] }
70 Set 13: { [F -> (E).] }
71 Set 14: { [E' -> +T.E'], [E' -> .+TE'], [E' -> .-TE'], [E' -> .] }
    | read '+' -> set 7, '-' -> set 8
72 Set 15: { [E' -> -T.E'], [E' -> .+TE'], [E' -> .-TE'], [E' -> .] }
    | read '+' -> set 7, '-' -> set 8
73 Set 16: { [T' -> *F.T'], [T' -> .*FT'], [T' -> ./FT'], [T' -> .] }
    | read '*' -> set 10, '/' -> set 11
74 Set 17: { [T' -> /F.T'], [T' -> .*FT'], [T' -> ./FT'], [T' -> .] }
    | read '*' -> set 10, '/' -> set 11
75 Set 18: { [E' -> +TE'.] }
76 Set 19: { [E' -> -TE'.] }
77 Set 20: { [T' -> *FT'.] }
78 Set 21: { [T' -> /FT'.] }
79 }
80

```

81 SLR(1) Table:

82	State	action										
		goto										
83		\$	+	-	*	/	()				
	num	E	T	F	E'	T'						
84	0							S1	S2			
		3	4	5								
85	1							S1	S2			
		6	4	5								
86	2	R9	R9	R9	R9	R9	R9					
87	3	ACC										
88	4	R3	S7	S8						R3		
89	5	R7	R7	R7	S10	S11	R7					
								12				
90	6							S13				
91	7							S1	S2			
		14	5									
92	8							S1	S2			
		15	5									
93	9	R0	R0									
94	10							S1	S2			
		16										

95	11						S1	S2
				17				
96	12		R4	R4	R4			R4
97	13		R8	R8	R8	R8	R8	R8
98	14		R3	S7	S8			R3
					18			
99	15		R3	S7	S8			R3
					19			
100	16		R7	R7	R7	S10	S11	R7
					20			
101	17		R7	R7	R7	S10	S11	R7
					21			
102	18		R1					R1
103	19		R2					R2
104	20		R5	R5	R5			R5
105	21		R6	R6	R6			R6

```

106
107
108 Starting analysis:
109 Step 1:
110 { <0, > }
111 (num+num)*num$
112 shift 1
113
114 Step 2:
115 { <0, >, <1, (> }
116 num+num)*num$
117 shift 2
118
119 Step 3:
120 { <0, >, <1, (>, <2, num> }
121 +num)*num$
122 reduce by: F -> num, goto[1,F]=5
123
124 Step 4:
125 { <0, >, <1, (>, <5, F> }
126 +num)*num$
127 reduce by: T' -> ε, goto[5,T']=12
128
129 Step 5:
130 { <0, >, <1, (>, <5, F>, <12, T'> }
131 +num)*num$
132 reduce by: T -> FT', goto[1,T]=4
133
134 Step 6:
135 { <0, >, <1, (>, <4, T> }
136 +num)*num$
137 shift 7
138
139 Step 7:
140 { <0, >, <1, (>, <4, T>, <7, +> }
141 num)*num$

```



```

142 shift 2
143
144 Step 8:
145 { <0, >, <1, (>, <4, T>, <7, +>, <2, num> }
146 )*num$
147 reduce by: F -> num, goto[7,F]=5
148
149 Step 9:
150 { <0, >, <1, (>, <4, T>, <7, +>, <5, F> }
151 )*num$
152 reduce by: T' -> ε, goto[5,T']=12
153
154 Step 10:
155 { <0, >, <1, (>, <4, T>, <7, +>, <5, F>, <12, T'> }
156 )*num$
157 reduce by: T -> FT', goto[7,T]=14
158
159 Step 11:
160 { <0, >, <1, (>, <4, T>, <7, +>, <14, T> }
161 )*num$
162 reduce by: E' -> ε, goto[14,E']=18
163
164 Step 12:
165 { <0, >, <1, (>, <4, T>, <7, +>, <14, T>, <18, E'> }
166 )*num$
167 reduce by: E' -> +TE', goto[4,E']=9
168
169 Step 13:
170 { <0, >, <1, (>, <4, T>, <9, E'> }
171 )*num$
172 reduce by: E -> TE', goto[1,E]=6
173
174 Step 14:
175 { <0, >, <1, (>, <6, E> }
176 )*num$
177 shift 13
178
179 Step 15:
180 { <0, >, <1, (>, <6, E>, <13, )> }
181 *num$
182 reduce by: F -> (E), goto[0,F]=5
183
184 Step 16:
185 { <0, >, <5, F> }
186 *num$
187 shift 10
188
189 Step 17:
190 { <0, >, <5, F>, <10, *> }
191 num$
192 shift 2
193
194 Step 18:
195 { <0, >, <5, F>, <10, *>, <2, num> }
196 $
197 reduce by: F -> num, goto[10,F]=16
198
199 Step 19:

```

```

200 { <0, >, <5, F>, <10, *>, <16, F> }
201 $
202 reduce by: T' -> ε, goto[16,T']=20
203
204 Step 20:
205 { <0, >, <5, F>, <10, *>, <16, F>, <20, T'> }
206 $
207 reduce by: T' -> *FT', goto[5,T']=12
208
209 Step 21:
210 { <0, >, <5, F>, <12, T'> }
211 $
212 reduce by: T -> FT', goto[0,T]=4
213
214 Step 22:
215 { <0, >, <4, T> }
216 $
217 reduce by: E' -> ε, goto[4,E']=9
218
219 Step 23:
220 { <0, >, <4, T>, <9, E'> }
221 $
222 reduce by: E -> TE', goto[0,E]=3
223
224 Step 24:
225 { <0, >, <3, E> }
226 $
227 ACC

```

5. LL1synthesis

```

1 fschi@FCXiaoXin:/mnt/v/Code/exp_compile/2_yacc/parse$
  ./build/LL1synthesis
2 Input file: data/expression.in
3
4 Start nonrecursive analysis.
5 Stack: E $      Input: ( num + num * num ) * num $      Output: E -
  > T E'
6 Stack: T E' $   Input: ( num + num * num ) * num $      Output: T -
  > F T'
7 Stack: F T' E' $      Input: ( num + num * num ) * num $
  Output: F -> ( E )
8 Stack: ( E ) T' E' $   Input: ( num + num * num ) * num $
  eliminate (
9 Stack: E ) T' E' $     Input: num + num * num ) * num $
  Output: E -> T E'
10 Stack: T E' ) T' E' $   Input: num + num * num ) * num $
  Output: T -> F T'
11 Stack: F T' E' ) T' E' $      Input: num + num * num ) * num $
  Output: F -> num
12 Stack: num T' E' ) T' E' $     Input: num + num * num ) * num $
  eliminate num
13 Stack: T' E' ) T' E' $   Input: + num * num ) * num $      Output: T'
  -> ε

```

```

14 Stack: E' ) T' E' $      Input: + num * num ) * num $      Output: E'
   -> + T E'
15 Stack: + T E' ) T' E' $      Input: + num * num ) * num $
   eliminate +
16 Stack: T E' ) T' E' $      Input: num * num ) * num $      Output: T -
   > F T'
17 Stack: F T' E' ) T' E' $      Input: num * num ) * num $
   Output: F -> num
18 Stack: num T' E' ) T' E' $      Input: num * num ) * num $
   eliminate num
19 Stack: T' E' ) T' E' $      Input: * num ) * num $      Output: T' -> * F
   T'
20 Stack: * F T' E' ) T' E' $      Input: * num ) * num $      eliminate *
21 Stack: F T' E' ) T' E' $      Input: num ) * num $      Output: F -
   > num
22 Stack: num T' E' ) T' E' $      Input: num ) * num $      eliminate
   num
23 Stack: T' E' ) T' E' $      Input: ) * num $      Output: T' -> ε
24 Stack: E' ) T' E' $      Input: ) * num $      Output: E' -> ε
25 Stack: ) T' E' $      Input: ) * num $      eliminate )
26 Stack: T' E' $      Input: * num $      Output: T' -> * F T'
27 Stack: * F T' E' $      Input: * num $      eliminate *
28 Stack: F T' E' $      Input: num $      Output: F -> num
29 Stack: num T' E' $      Input: num $      eliminate num
30 Stack: T' E' $      Input: $      Output: T' -> ε
31 Stack: E' $      Input: $      Output: E' -> ε
32 Stack: $      Input: $      eliminate $
33 Analysis succeed.

```

6. LRsynthesis

```

1 fschi@FCXiaoXin:/mnt/v/Code/exp_compile/2_yacc/parse$
  ./build/LRsynthesis
2 Input file: data/expression.in
3
4 Generating LR(0) item sets.
5 LR(0) item sets generated.
6
7 Starting analysis:
8 Step 1:
9 { <0, > }
10 (num+num*num)*num$
11 shift 1
12
13 Step 2:
14 { <0, >, <1, (> }
15 num+num*num)*num$
16 shift 2
17
18 Step 3:
19 { <0, >, <1, (>, <2, num> }
20 +num*num)*num$
21 reduce by: F -> num, goto[1,F]=5
22
23 Step 4:

```

```

24 { <0, >, <1, (>, <5, F> }
25 +num*num)*num$
26 reduce by: T -> F, goto[1,T]=4
27
28 Step 5:
29 { <0, >, <1, (>, <4, T> }
30 +num*num)*num$
31 reduce by: E -> T, goto[1,E]=6
32
33 Step 6:
34 { <0, >, <1, (>, <6, E> }
35 +num*num)*num$
36 shift 7
37
38 Step 7:
39 { <0, >, <1, (>, <6, E>, <7, +> }
40 num*num)*num$
41 shift 2
42
43 Step 8:
44 { <0, >, <1, (>, <6, E>, <7, +>, <2, num> }
45 *num)*num$
46 reduce by: F -> num, goto[7,F]=5
47
48 Step 9:
49 { <0, >, <1, (>, <6, E>, <7, +>, <5, F> }
50 *num)*num$
51 reduce by: T -> F, goto[7,T]=12
52
53 Step 10:
54 { <0, >, <1, (>, <6, E>, <7, +>, <12, T> }
55 *num)*num$
56 shift 9
57
58 Step 11:
59 { <0, >, <1, (>, <6, E>, <7, +>, <12, T>, <9, *> }
60 num)*num$
61 shift 2
62
63 Step 12:
64 { <0, >, <1, (>, <6, E>, <7, +>, <12, T>, <9, *>, <2, num> }
65 )*num$
66 reduce by: F -> num, goto[9,F]=14
67
68 Step 13:
69 { <0, >, <1, (>, <6, E>, <7, +>, <12, T>, <9, *>, <14, F> }
70 )*num$
71 reduce by: T -> T*F, goto[7,T]=12
72
73 Step 14:
74 { <0, >, <1, (>, <6, E>, <7, +>, <12, T> }
75 )*num$
76 reduce by: E -> E+T, goto[1,E]=6
77
78 Step 15:
79 { <0, >, <1, (>, <6, E> }
80 )*num$
81 shift 11

```

```

82
83 Step 16:
84 { <0, >, <1, (>, <6, E>, <11, )> }
85 *num$
86 reduce by: F -> (E), goto[0,F]=5
87
88 Step 17:
89 { <0, >, <5, F> }
90 *num$
91 reduce by: T -> F, goto[0,T]=4
92
93 Step 18:
94 { <0, >, <4, T> }
95 *num$
96 shift 9
97
98 Step 19:
99 { <0, >, <4, T>, <9, *> }
100 num$
101 shift 2
102
103 Step 20:
104 { <0, >, <4, T>, <9, *>, <2, num> }
105 $
106 reduce by: F -> num, goto[9,F]=14
107
108 Step 21:
109 { <0, >, <4, T>, <9, *>, <14, F> }
110 $
111 reduce by: T -> T*F, goto[0,T]=4
112
113 Step 22:
114 { <0, >, <4, T> }
115 $
116 reduce by: E -> T, goto[0,E]=3
117
118 Step 23:
119 { <0, >, <3, E> }
120 $
121 ACC

```

7. calc (Yacc)

1. test.p

```

1 1b-1.2E-2 + 3
2 (1.0+3.1415)*( 13.3/.5e-2)
3 1.14514+ 4.15411* 35768*(1.43+9.53)
4

```

2. 测试输出

```

1 fschi@FCXiaoXin:/mnt/v/Code/exp_compile/2_yacc/yacc$ ./calc <
  test.p
2 Reduce by frac -> num

```

```
3  bReduce by frac -> num
4  Reduce by exp -> exp - term
5  Reduce by frac -> num
6  Reduce by exp -> exp + term
7  Ans = 3.988000
8
9  Reduce by frac -> num
10 Reduce by frac -> num
11 Reduce by exp -> exp + term
12 Reduce by frac -> (expression)
13 Reduce by frac -> num
14 Reduce by frac -> num
15 Reduce by term -> term / num
16 Reduce by frac -> (expression)
17 Reduce by term -> term * num
18 Ans = 11016.390000
19
20 Reduce by frac -> num
21 Reduce by frac -> num
22 Reduce by frac -> num
23 Reduce by term -> term * num
24 Reduce by frac -> num
25 Reduce by frac -> num
26 Reduce by exp -> exp + term
27 Reduce by frac -> (expression)
28 Reduce by term -> term * num
29 Reduce by exp -> exp + term
30 Ans = 1628484.048161
```

源程序与可执行文件

1. 方法1-3详见 `./parser`
2. 方法4详见 `./yacc`