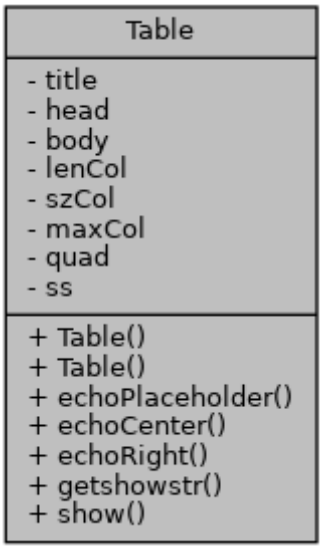
小型编译器框架构建与PL/0语言解释器实例化

——设计说明书

nekko

类设计

Table

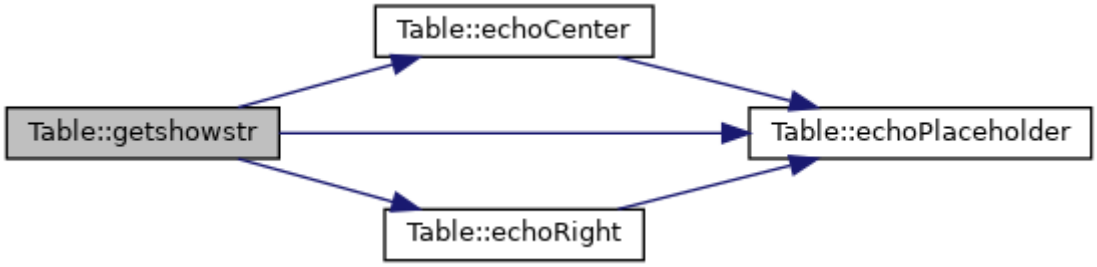


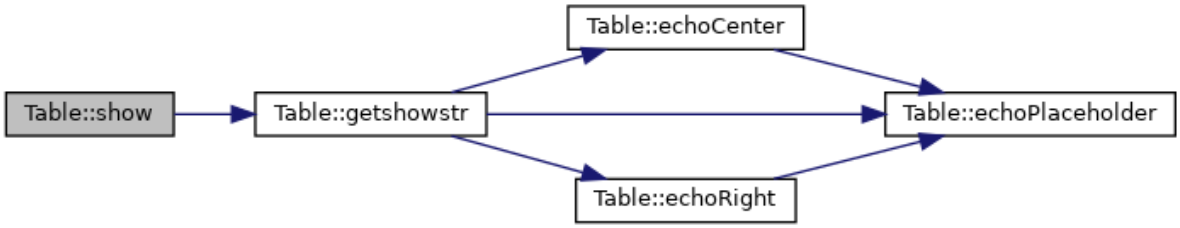
用于对输出的表单信息进行美化(purify)输出。

成员函数

	Table ()
	Table (string **title**, vector< string > &**head**, vector< vector< string > > &**body**, int **szCol**=64, int **quad**=3)
void	**echoPlaceholder** (char c, int times, bool nextLine=false)
void	**echoCenter** (string str, bool nextLine=false, int mxl=-1)
void	**echoRight** (string str, bool nextLine=false, int mxl=-1)
string	**getshowstr** (bool showCnt=true)
void	**show** (bool showCnt=true)

- `echoPlaceholder` 用于输出占位符。
- `echoCenter/echoRight` 用于对齐数据。
- `getshowstr` 用于生成格式化后的数据字符串。
- `show` 用于向控制台输出数据字符串。



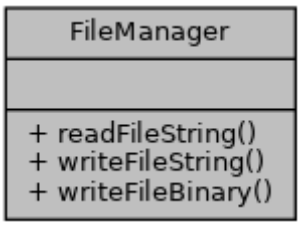


成员变量

string	**title**
vector< string >	**head**
vector< vector< string > >	**body**
vector< int >	**lenCol**
int	**szCol**
int	**maxCol**
int	**quad**
stringstream	**ss**

- `title` 表示数据表单的表头。
- `head` 表示数据表单的列标题。
- `body` 表示数据表单的数据体。
- `lenCol` 列数据个数。
- `szCol` 列块大小。
- `maxCol` 列最大长度。
- `quad` 占位符长度。
- `ss` 存储格式化后数据的字符串流。

FileManager



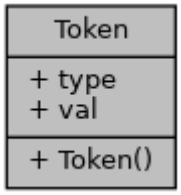
用于处理文件读写。

静态成员函数

static string	**readFileString** (string filename)
static void	**writeFileString** (string filename, string data)
static void	**writeFileBinary** (string filename, char *buffer)

- `readFileString` 以字符串形式读取文件。
- `writeFileString` 以字符串形式写入文件。

Token



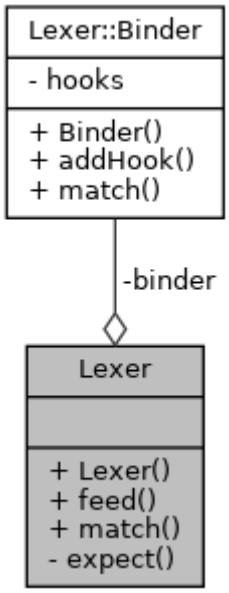
用于存储 `Lexer` 分词结果。

成员变量

TokenType	**type**
string	**val**

- `type` Token类型。
- `val` 分词时匹配的原始字符串。

Lexer



用于处理分词过程，本质上是一个正则库。

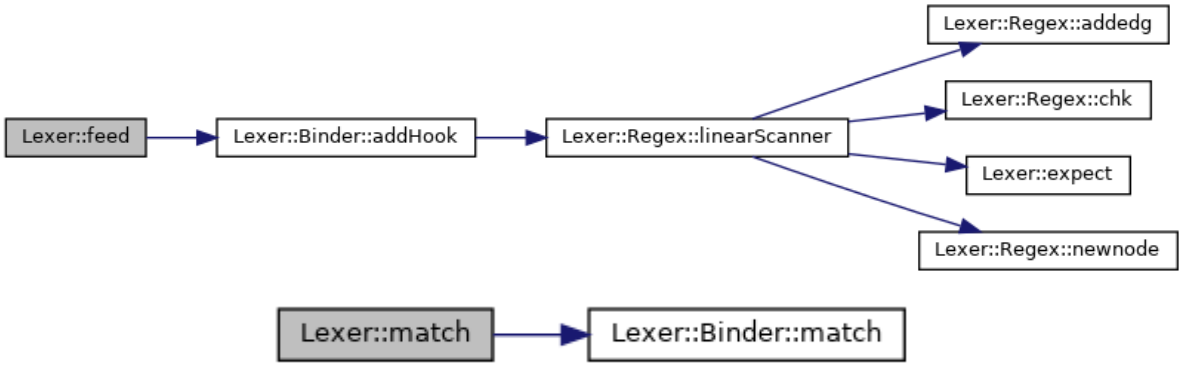
由多个子类组成：

class	**Automachine**
class	**Binder**
class	**DFA**
class	**NFA**
class	**Regex**

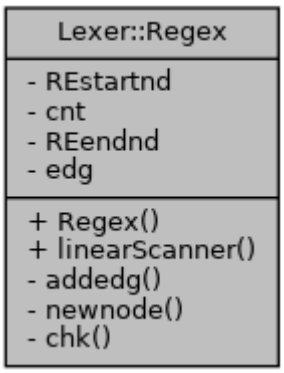
成员函数

	Lexer ()
Lexer &	**feed** (string reg, function< void(string)> act)
Lexer &	**match** (string str)
static void	**expect** (bool flag)

- `feed` 通过传入 `<正则表达式，回调函数>`，来对词法模板进行预编译。
- `match` 通过传入源程序，来对源程序进行正则匹配，并通过回调函数处理匹配结果。
- `expect` 相当于断言(assert)函数，如果 `flag` 为假，则终止程序。



Lexer::Regex



用于正则表达式预编译部分的预处理（使用 Thompson 方法构造正规语言的 NFA）。

成员变量

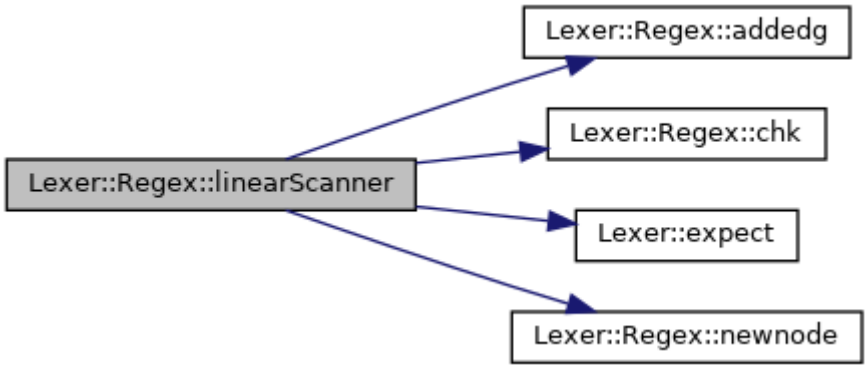
int	REstartnd
int	cnt
set< int >	REendnd
vector< Edge >	edg

- `REstartnd` 表示构造出的 NFA 的起始节点。
- `cnt` 表示构造出的 NFA 的节点个数。
- `edg` 表示构造出的 NFA 的转移边。

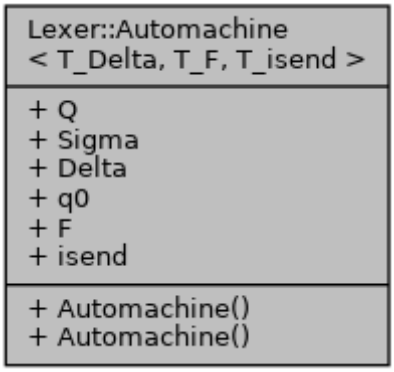
成员函数

	Regex ()
void	linearScanner (string str)
void	addedg (int u, int v, int w)
int	newnode ()
bool	chk (char c)

- `linearScanner` 通过线性扫描，将正则表达式进行分词、解析、构造对应的 NFA。
- `addedg` 用于添加新的转移边。
- `newnode` 用于新建 NFA 节点。
- `chk` 用于在线性扫描时判断是否是合法字符。



Lexer::Automachine



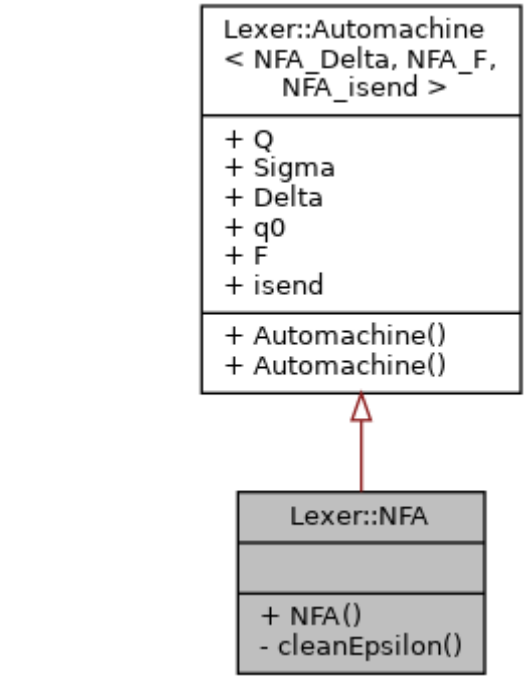
NFA和DFA的基类，定义自动机的基本模式。

成员变量

int	Q
int	Sigma
T_Delta	Delta
int	q0
T_F	F
T_isend	isend

- `Q` 表示最大状态数。
- `Sigma` 表示字符集大小（最大字符编码）。
- `Delta` 表示状态转移函数（通过 `nextState = Delta[currentState][currentChar]` 进行转移）。
- `q0` 表示初始状态。
- `F` 表示终止状态集合。
- `isend` 表示状态是否是终止状态，即 $x \in F \Leftrightarrow \text{isend}(x) = \text{true}$。

Lexer::NFA



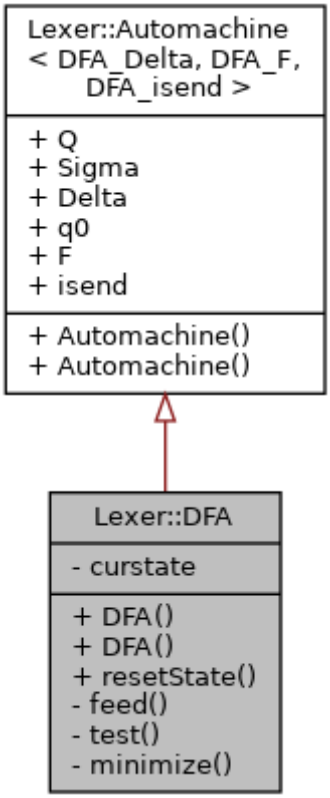
非确定型有限自动机[2]，用于存储经过 Topmost 方法构造成 NFA 的正则表达式。

成员函数

void	cleanEpsilon ()

- `cleanEpsilon` 清空 NFA 中的 ε 边，将 $\varepsilon - NFA$ 定型化为 $NFA_{\varepsilon-free}$

Lexer::DFA

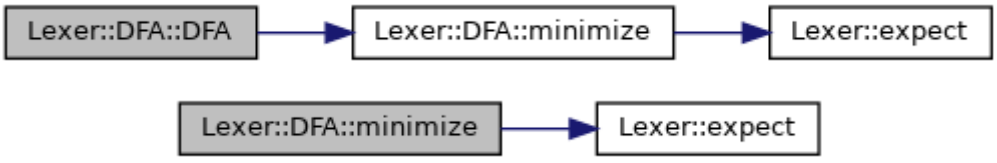


确定型有限自动机[3]，用于将 NFA 进行确定化以及最小化DFA，以及执行一部分的正则匹配功能。

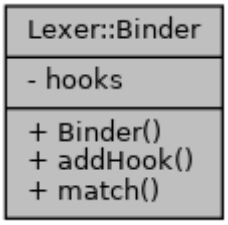
成员函数

	DFA (int **Q**, int **Sigma**, **DFA_Delta** **Delta**, int **q0**, **DFA_F** **F**, **DFA_isend** **isend**)
	DFA (**NFA** &nfa)
void	**resetState** ()
bool	**feed** (char c)
bool	**test** (char c)
void	**minimize** ()

- `resetState` 用于重置正则匹配时的当前状态节点。
- `feed` 用于转移正则匹配时的当前状态节点。
- `test` 用于正则匹配是判断是否可以转移（`Delta` 中有出边）。
- `minimize` 用于最小化确定型有限自动机。



Lexer::Binder



用于绑定正则表达式和回调函数。

成员变量

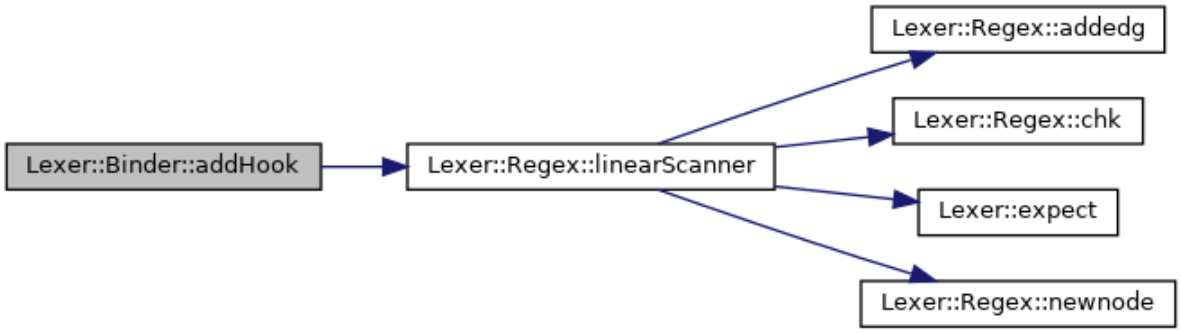
typedef tuple< string, function< void(string)>, DFA > regexHook
vector< regexHook > hooks

- `hooks` 用于存储绑定的正则表达式、回调函数和预编译得到的 DFA，其中 `string` 部分也充当回填与目标串匹配上的部分。

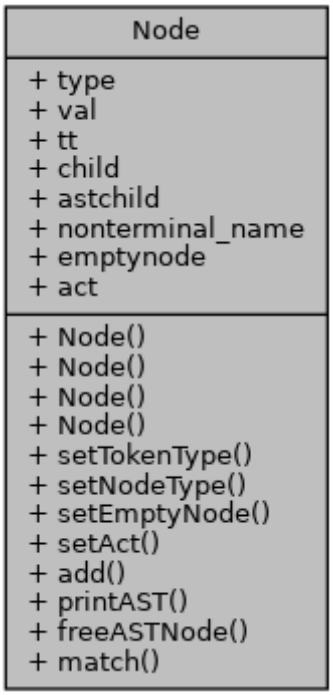
成员函数

	Binder ()
void	addHook (string reg, function< void(string)> act)
void	match (string str)

- `addHook` 用于添加一组绑定的正则表达式和回调函数。
- `match` 用于预编译所有正则串，以及匹配输入字符串 `str`。



Node



用于存储 CFG[4] 和 AST[5]。

成员变量

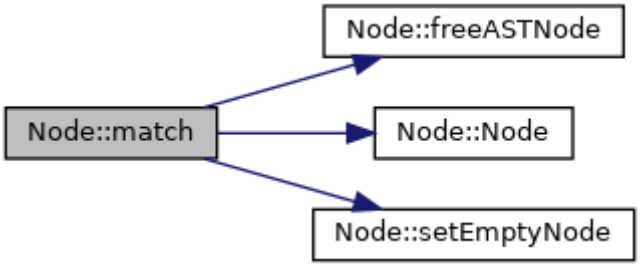
NodeType	type
string	val
TokenType	tt
vector< pair< vector< Node * >, F_ACT > >	child
vector< Node * >	astchild
string	nonterminal_name
bool	emptynode
FF_ACT	act

- `type` 表示节点类型，分为中间节点(Nonterminal)和终止节点(Terminal)。
- `val` 表示终止节点的匹配字符串。
- `tt` 表示终止节点的匹配Token类型。
- `child` 表示CFG存储的子节点（F_ACT表示CFG中间节点的特定规则匹配成功后的回调函数）。
- `astchild` 表示AST中存储的子节点。
- `nonterminal_name` 表示CFG的中间节点的符号名。
- `emptynode` 表示AST上的节点是否是空节点（即子树中不包含非空终止节点）。
- `act` 表示CFG中间节点匹配成功后的回调函数。

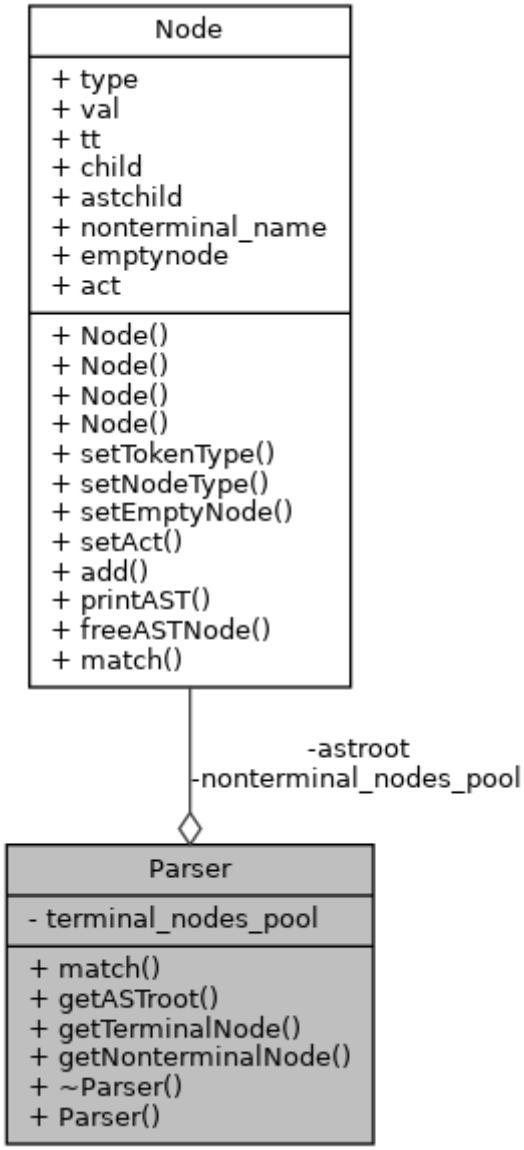
成员函数

	Node ()
	Node (NodeType type, string val, TokenType tt)
	Node (NodeType type, string val)
	Node (NodeType type)
Node *	setTokenType (TokenType tt)
Node *	setNodeType (NodeType type)
Node *	setEmptyNode (bool flag)
Node *	setAct (FF_ACT act)
Node *	add (vector< Node * > args, F_ACT act=NULL)
static string	printAST (Node *nd)
static void	freeASTNode (Node *nd)
static Node *	match (Node *S)

- `setTokenType` 设置节点的Token类型。
- `setNodeType` 设置节点的Node类型。
- `setEmptyNode` 设置节点的空类型。
- `setAct` 设置节点的匹配回调函数。
- `add` 用于CFG添加节点规则（使用LL(1)[6]文法）。
- `printAST` 用于输出抽象语法树(AST)。
- `freeASTNode` 用于内存回收。
- `match` 用于使用CFG进行匹配tokens，这里使用的算法是**递归下降子程序法**(RDP[7])。



Parser



用于语法分析和语义分析。

成员变量

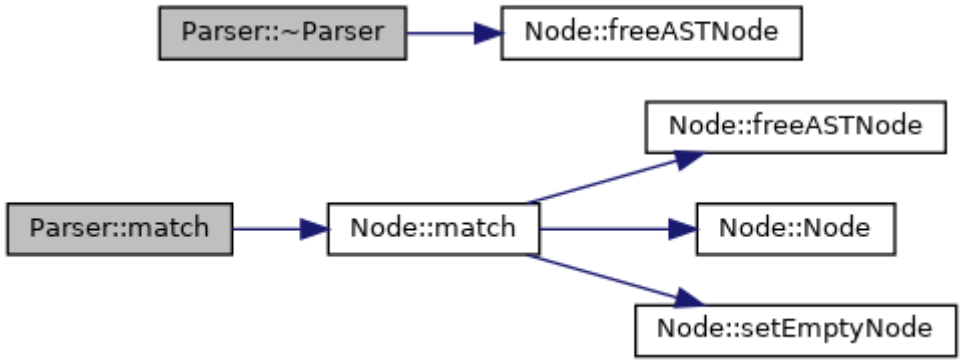
Node *	nonterminal_nodes_pool
vector< Node * >	terminal_nodes_pool
Node *	astroot

- `nonterminal_nodes_pool` 存储CFG的中间节点。
- `terminal_nodes_pool` 存储终止节点。
- `astroot` 存储构建的抽象语法树(AST)的根节点。

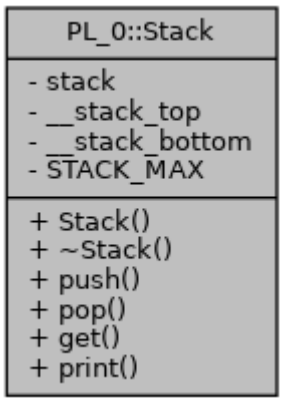
成员函数

Node *	match (Node *S)
Node *	getASTroot ()
Node *	getTerminalNode (string val)
Node *	getNonterminalNode (int idx)
	~Parser ()
	Parser ()

- `match` 调用 `Node` 中的 `match` 进行语法分析。
- `getASTroot` 返回 `astroot`。
- `getTerminalNode` 通过终止节点的值返回一个终止节点。
- `getNonterminalNode` 通过中间节点的编码返回对应的中间节点。



PL_0::Stack



用于Virtual Machine中存储局部变量的栈空间。

成员变量

int *	stack
int *	__stack_top
int *	__stack_bottom

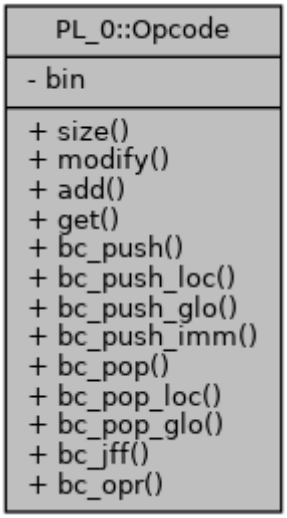
- `stack` 存储栈空间。
- `__stack_top` 表示栈顶（下标）。
- `__stack_bottom` 表示栈底（下标）。

成员函数

```
        Stack (int *st, int *sb, int stack_max=STACK_MAX)
        ~Stack ()
void    push (int x)
 int    pop ()
int &   get (int idx)
void    print (int l, int r)
```

- `push` 将 `x` 压入栈。
- `pop` 弹出栈顶元素，并返回它得值。
- `get` 获取相对于栈底 `idx` 的位置的值（用于读写局部变量）。
- `print` 输出栈数组下标在 `[l,r]` 中的元素的值（用于调试）。

PL_0::Opcode



表示、翻译和存储字节码。

成员变量

```
enum class   BasicCommand {
               push = 0 , pop , jff , plus ,
               minus , multi , div , odd ,
               eq , neq , lt , le ,
               gt , ge , inp , out
             }
vector< vector< int > >   bin
```

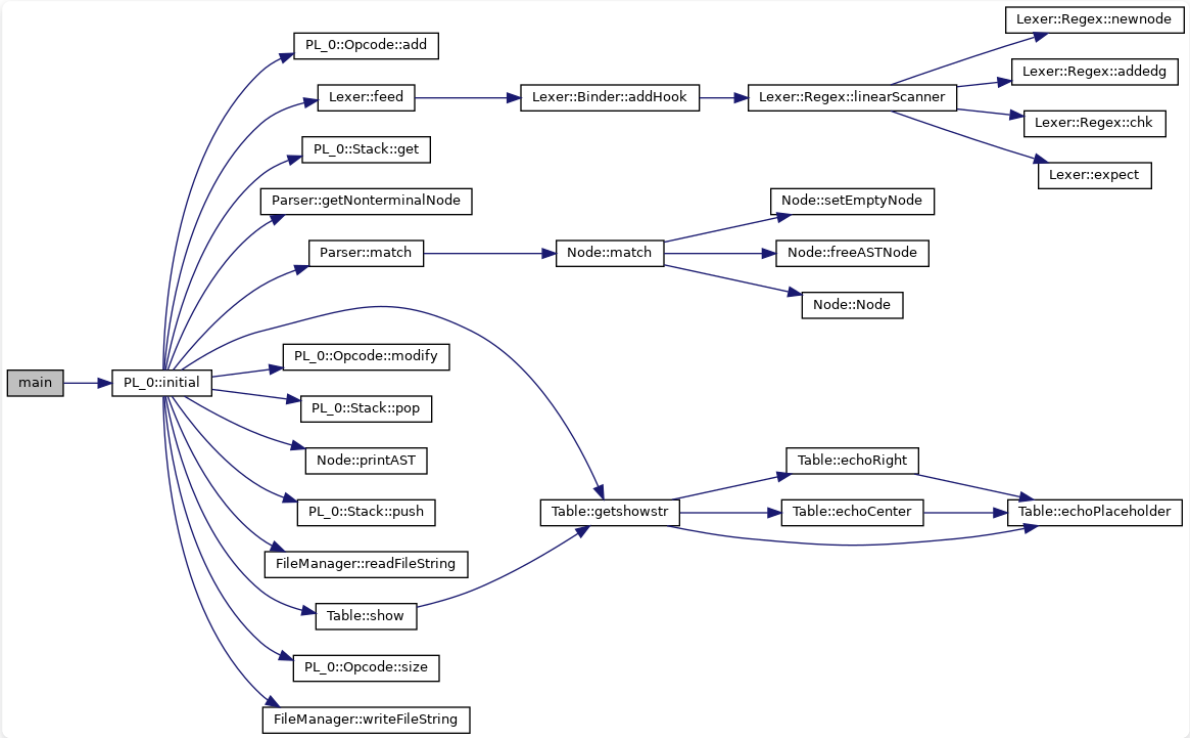
- `BasicCommand` 表示各类型字节码。
- `bin` 表示总生成的字节码。

成员函数

int	**size** () const
Opcode &	**modify** (int idx, vector< int > odt)
Opcode &	**add** (vector< int > odt)
vector< int >	**get** (int cnt) const
static vector< int >	**bc_push** (int ty, int data)
static vector< int >	**bc_push_loc** (int data)
static vector< int >	**bc_push_glo** (int data)
static vector< int >	**bc_push_imm** (int data)
static vector< int >	**bc_pop** (int ty, int data)
static vector< int >	**bc_pop_loc** (int data)
static vector< int >	**bc_pop_glo** (int data)
static vector< int >	**bc_jff** (int data)
static vector< int >	**bc_opr** (string op)

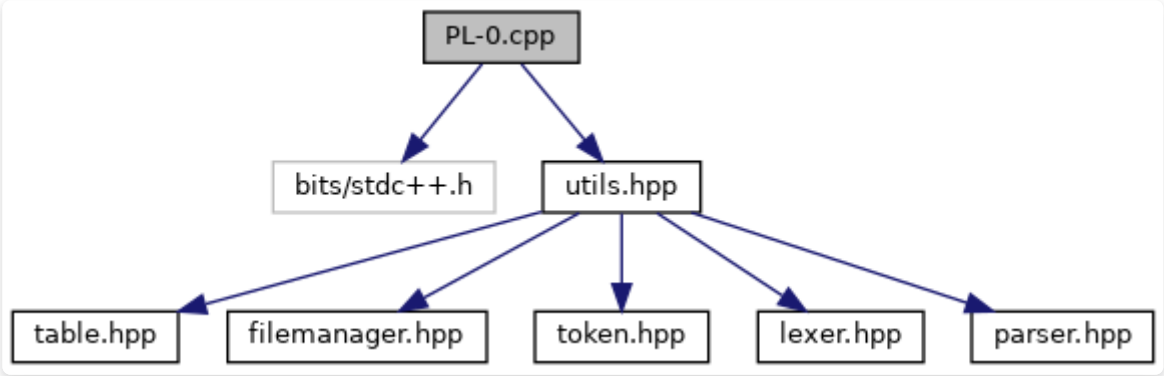
- `size` 返回当前字节码总大小。
- `modify` 修改某一地址的字节码（用于回填跳转地址）。
- `add` 添加一条字节码。
- `get` 获取某条字节码（用于汇总输出表单）。
- `bc_*` 依次表示填充各类型字节码。

函数调用关系总览图



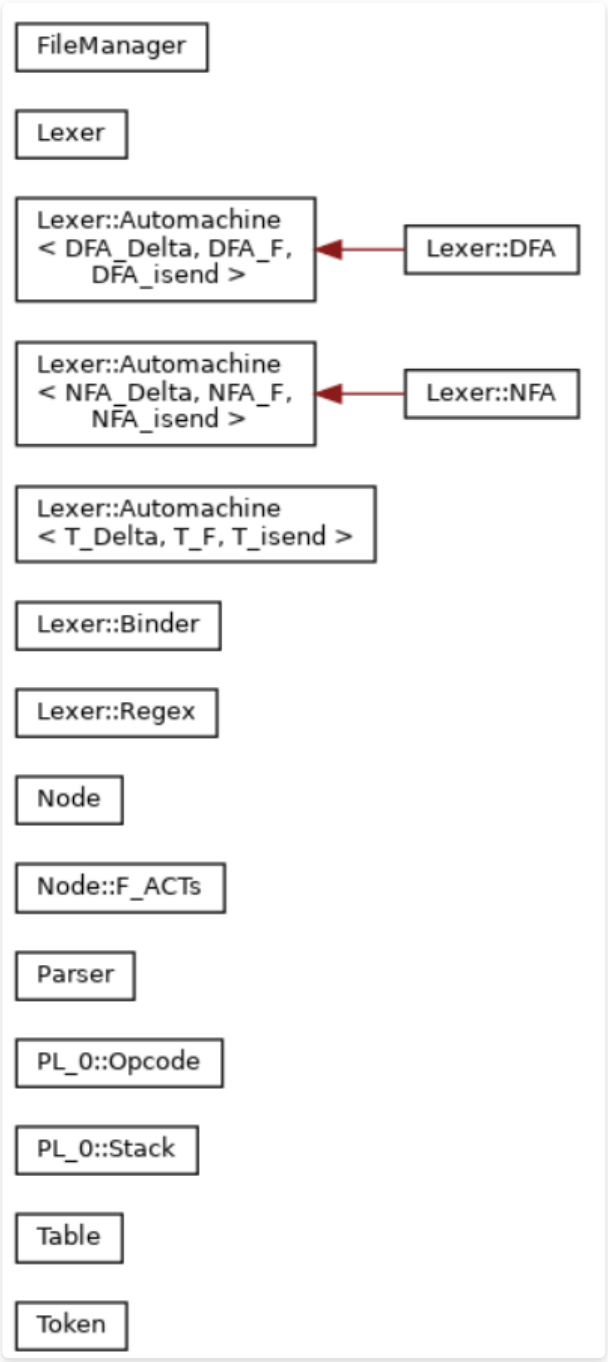
高内聚，低耦合

文件调用关系总览图



高内聚，低耦合

类关系总览图

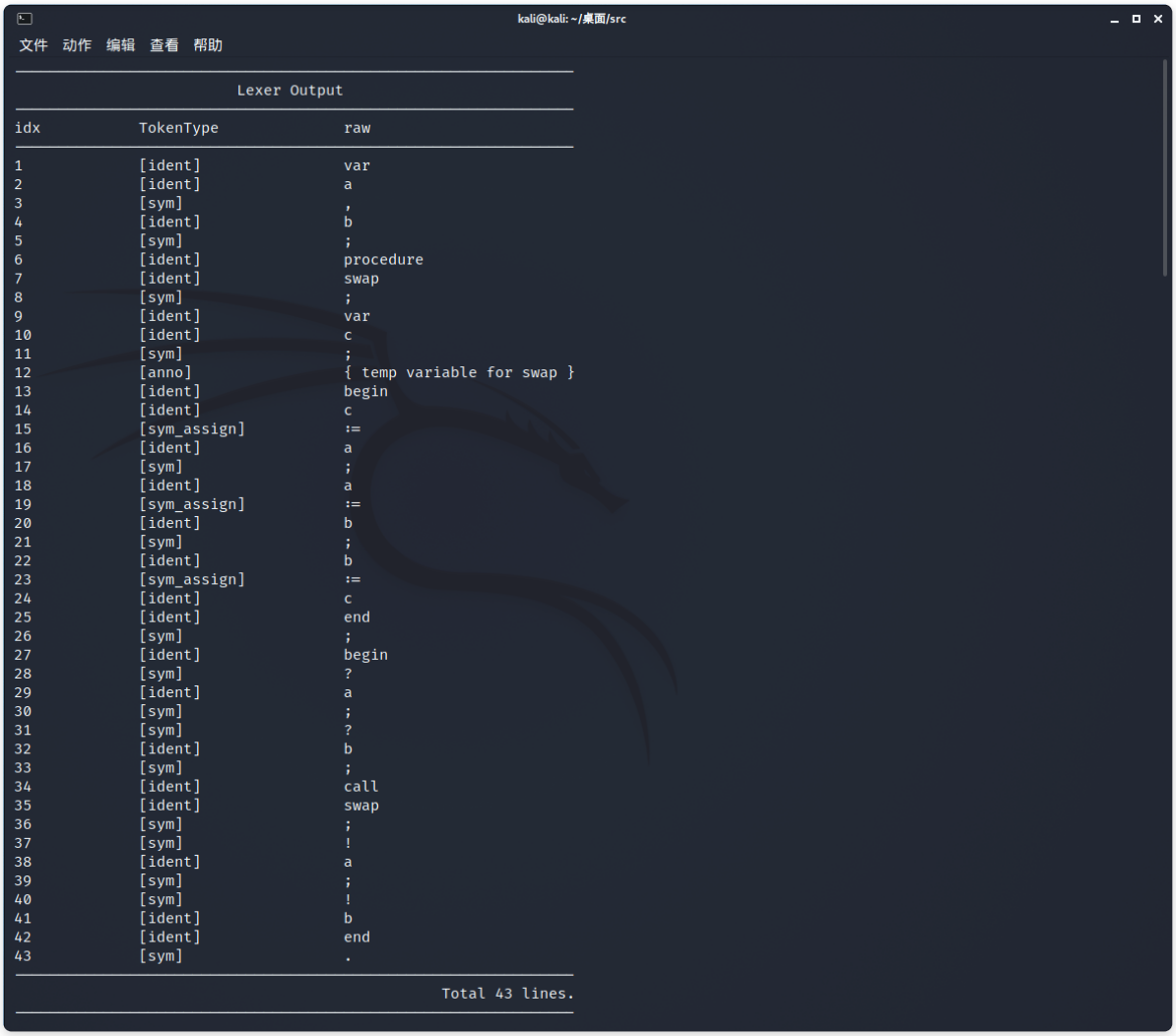


高内聚，低耦合

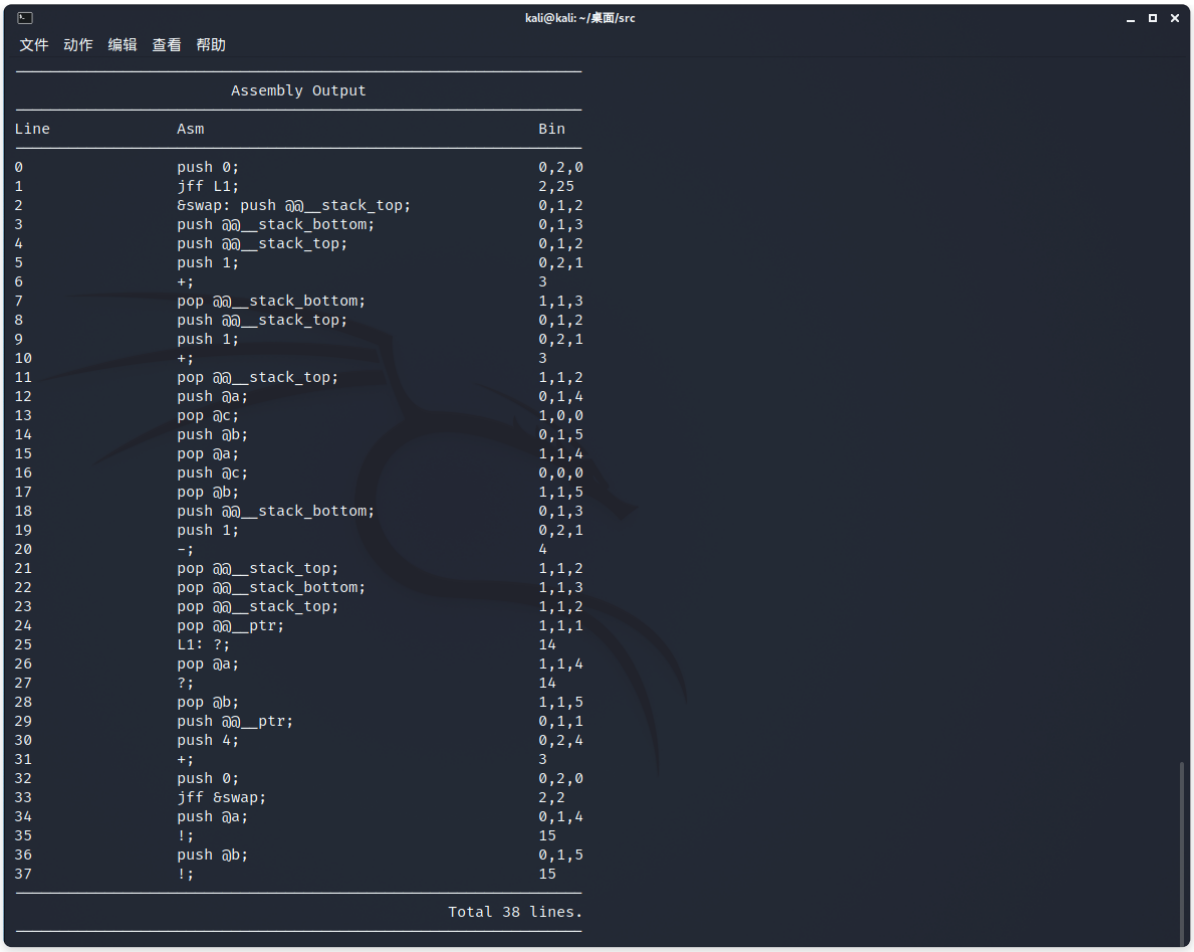
算法设计及技术难点与实现方案

表单输出美化

直接输出数据表不够优美，这里通过 `table` 类来对表单输出进行了美化。



Lexer Output



Assembly Output

技术难点与实现方法

通过计算每一列的最大宽度，就可以计算出对齐所需空白占位符长度。

PL/0语言设计总览

类别	词法	符号	说明
关键字	begin	beginsym	块开始
	end	endsym	块结束
	call	callsym	调用函数
	const	constsym	常量声明
	var	varsym	变量声明
	if	ifsym	判断语句
	then	thensym	判断语句
	while	whilesym	循环语句
	do	dosym	循环语句
	procedure	proceduresym	过程声明
	?	readsym	读入变量
	!	writesym	输出表达式
标识符	[ident]	ident	标识符
字面量	[literal]	literal	字面量（整数）
运算符	+	plus	加
	-	minus	减
	*	times	乘
	/	slash	整除
	odd	oddsym	判断奇数
	=	eql	等于
	#	neq	不等于
	<	lss	小于
	<=	leq	小于等于
	>	gtr	大于
	>=	geq	大于等于
	:=	becomes	赋值
界符	(	lparan	左括号
	)	rparan	右括号
	,	comma	逗号
	;	semicolon	分号

类别	词法	符号	说明
	.	period	点

设计Token的正则表达式

- 数字：`[0-9][0-9]*`
- 标识符：`[_a-zA-Z][_a-zA-Z0-9]*`
- 赋值符号：`\:\=`
- 小于等于符号：`\<\=`
- 大于等于符号：`\>\=`
- 其它符号：`[!?:;.,#<=>+-\*/\(\)]`
- 注释：`\{[\f\n\r\t\v -z|~]*\}`
- 空白字符：`[ \f\n\r\t\v][ \f\n\r\t\v]*`

设计自动机

一个自动机通过五元组 $(Q, \Sigma, \delta, q_0, F)$ 表示，其中 Q 表示状态集，Σ 表示字符集，$\delta(q, c)$ 表示二元转移函数（从状态 q 通过转移字符 c 所到达的状态（集）），q_0 表示起始状态，F 表示终止状态集。

为了简便起见，记 $\hat{\delta}(q, cw) = \hat{\delta}(\delta(q, c), w), (w \in \Sigma^*)$。

技术难点与实现方案

容易观察到，DFA和NFA都属于自动机，但它们的转移函数的定义不同（NFA为 `vector<vector<set<int>>>`，DFA为 `vector<vector<int>>`）。

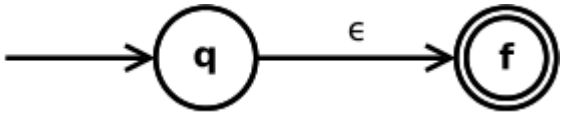
可以考虑使用模板类进行不同类型继承。

```
template<typename T_Delta, typename T_F, typename T_isend>
class Automachine {
    public:
    int Q; // node count (index from 1)
    int Sigma; // alphabet size
    T_Delta Delta; // δ(q,w)
    int q0; // start state(node)
    T_F F; // finished state
    T_isend isend; // marked if is finished
    Automachine() {}
    Automachine(int Q, int Sigma, T_Delta Delta, int q0, T_F F, T_isend isend) :
Q(Q), Sigma(Sigma), Delta(Delta), q0(q0), F(F), isend(isend) {};
};
class NFA : private Automachine<NFA_Delta, NFA_F, NFA_isend>; // NFA = (Q, Σ, δ,
q0, F)
class DFA : private Automachine<DFA_Delta, DFA_F, DFA_isend>; // DFA = (Q, Σ, δ,
q0, F)
```

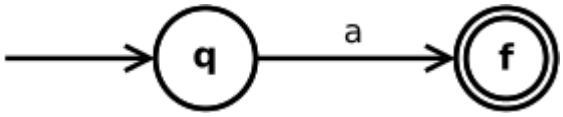
从正则表达式到ε-NFA（Thompson构造法[8]）

通过下列转化，可以将正则表达式构造成等价的 $\varepsilon - NFA$。

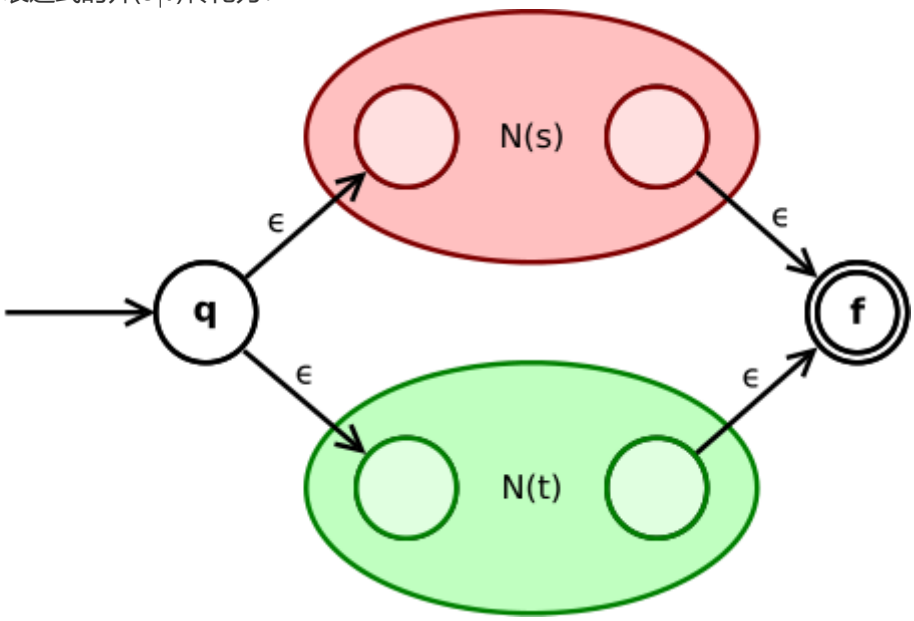
- 空表达式 ε 转化为：



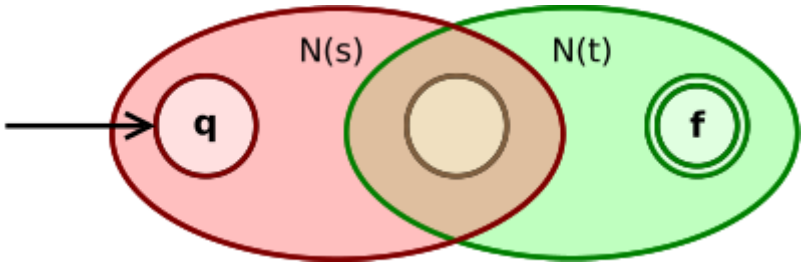
- 单个符号 a 转化为：



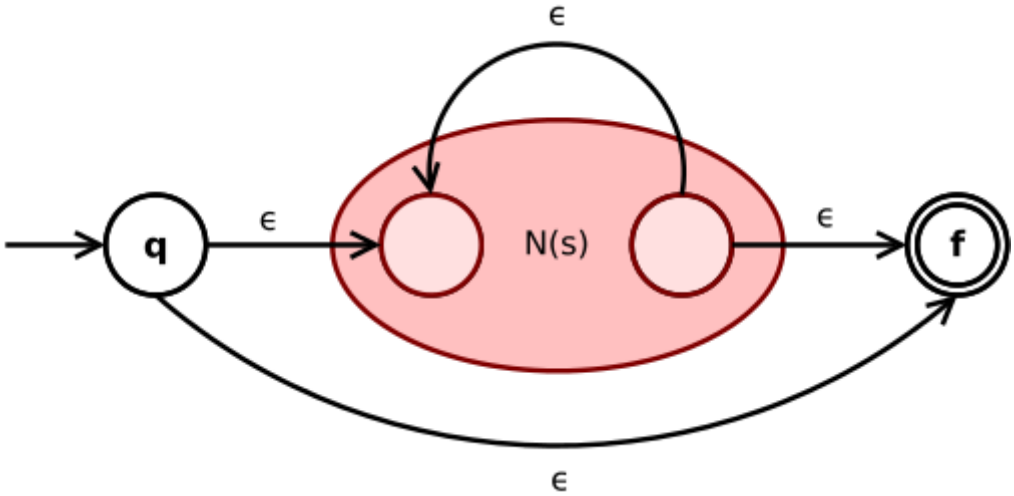
- 表达式的并$(s|t)$转化为：



- 表达式的连接(st)转化为：



- Kleen闭包(s^*)转化为：



从ε-NFA到NFA（化简ε-NFA）

记 $\mathrm{E}(q)=\{p|\hat{\delta}(q,\varepsilon)=p\}$，表示状态 q 只沿 ε 边所能到达的状态集合。

则 $\forall c\in\Sigma,\delta_{NFA}(E(q),c)=\{\cup_{p\in S}E(p)|S=\{r|\delta(q,c)=r\}\}$。

实际上，就是将朴素模拟 $\varepsilon-NFA$ 的过程用简化状态集表示了。

从NFA到DFA（确定化NFA）

这里采用**子集构造**[8]的方法。

和化简 $\varepsilon-NFA$ 的步骤类似，记 $\Gamma_c(q)=\{p|\delta(q,c)=p\}$，设计如下 DFA：起始状态为 $\{q_0\}$，状态转移函数 $\delta_{DFA}(S,c)=\cup_{p\in S}\Gamma_c(p)$。

技术难点与实现方案

1. 对状态集合的映射，可以采用哈希法，也可以采用 `map<set<int>, int>` 存储（`set<int>` 可以作为 `map` 的键值）。
2. 通过子集构造方法确定化NFA，这里使用的是**广度优先搜索**(bfs[9])算法，即通过 `queue<set<int>> que;` 来存储状态集合 S，每次通过取队列首端元素进行更新。

从DFA到MFA（最小化DFA[10]）

最小化DFA有许多算法，这里选用比较直接的**填表算法**。

定义 p,q 状态相同，当且仅当 $\forall w\in\Sigma^+,s.t.\hat{\delta}(p,w)=\hat{\delta}(q,w)$，换而言之：若 $\exists w\in\Sigma^+,s.t.\hat{\delta}(p,w)\neq\hat{\delta}(q,w)$，则 p,q 状态不相同。

首先，若 $p\in F,q\notin F$，则 p,q 不同；其次，若 $\delta(p,c)=p',\delta(q,c)=q'$ 且 p',q' 不同，则 p,q 不同；同时，若 p,q 不同，则存在串 w，使得 $\hat{\delta}(p,w)\in F,\hat{\delta}(q,w)\notin F$ 或相反。

所以可以再次使用广度优先搜索算法，初始时将所有满足 $p\in F,q\neq F$ 的 (p,q) 点对加入队列，然后依次消除所有的不等价点对，最后剩下的点对就是所有的等价类。

之后只需要删除所有的不可从初始节点到达的节点就可以转化为MFA了；为了更容易实现字符串匹配，这里可以稍微破坏DFA的结构，即将所有不能到达终止节点的节点删除。

技术难点与实现方法

由于**填表算法**比较复杂，这部分的技术难点主要是编写与调试。

MFA匹配字符串

这里的技术难点主要分为如何利用预编译好的**DFA进行字符串匹配**，以及如何优美的**读入绑定结构**。

技术难点与实现方法

线性扫描字符串的时候，依次使用每一个DFA进行贪心匹配。

```
void match(string str) {
    int len = str.length(), ptr = 0;
    while(ptr < len) { // analyse raw code
        walka(&hk, hooks) {
            auto &raw = get<0>(hk);
            auto &act = get<1>(hk);
```

```
            auto &dfa = get<2>(hk);
            raw = "", dfa.resetState();
            int ptrmem = ptr, isendstate = 0;
            while(ptr < len && dfa.test(str[ptr])) {
                isendstate = dfa.feed(str[ptr]);
                raw += str[ptr ++];
            }
            if(isendstate) { // state in `endF`, and we catched a string!
                act(raw);
                goto nxt;
            } else {
                ptr = ptrmem;
            }
        }
        errorlog("error in `match`: can't match any regex, char: %c (ascii:
%d)", str[ptr], (int) str[ptr]);
        nxt: ;
    }
}
```

对于绑定匹配模式和匹配回调函数，这里通过 `Lexer& f() { return *this; }` 的方法，实现连续调用。

```
lexer.feed("[0-9][0-9]*",
           [&](string raw) {
               bpb("[number]", raw);
               tpb(TokenType :: number, raw);
           })
    .feed("[_a-zA-Z][_a-zA-Z0-9]*", // identifiers
          [&](string raw) {
              bpb("[ident]", raw);
              tpb(TokenType :: ident, raw);
          });
```

设计LL(1)文法

这里先给出EBNF范式的PL/0文法：

```
program = block "." ;
block = [ "const" ident "=" number {"," ident "=" number} ";"]
        [ "var" ident {"," ident} ";"]
        { "procedure" ident ";" block ";" } statement;
statement = [ ident ":=" expression | "call" ident
              | "?" ident | "!" expression
              | "begin" statement {";" statement } "end"
              | "if" condition "then" statement
              | "while" condition "do" statement ];
condition = "odd" expression |
            expression ("="|"#"|"<"|"<="|">"|">=") expression;
expression = [ "+"|"-"] term { ("+"|"-") term};
term = factor {("*"|"/") factor};
factor = ident | number | "(" expression ")";
```

由于Parser部分使用RDP实现，所以只需要去除**左递归**[11]即可（不一定需要完全达到LL(1)），只需要增加后缀部分拆分左递归式。

原始版本的PL/0对负数支持不太好，这里重写了部分定义。

```
<program> = <block> "."
<block> = <constdef> <vardef> <procdef> <statement>
<constdef>    = "const" <ident> "=" <number> <constdefpri> ";"
              | ""
<constdefpri> = "," <ident> "=" <number> <constdefpri>
              | ""
<vardef>    = "var" <ident> <vardefpri> ";"
            | ""
<vardefpri> = "," <ident> <vardefpri>
            | ""
<procdef> = "procedure" <ident> ";" <constdef> <vardef> <statement> ";"
<procdef>
          | ""
<statement>    = <ident> ":=" <expression>
               | "call" <ident>
               | "?" <ident>
               | "!" <expression>
               | "begin" <statement> <statementpri> "end"
               | "if" <condition> "then" <statement>
               | "while" <condition> "do" <statement>
<statementpri> = ";" <statement> <statementpri>
               | ""
<condition> = "odd" <expression>
            | <expression> "="  <expression>
            | <expression> "#"  <expression>
            | <expression> "<"  <expression>
            | <expression> "<=" <expression>
            | <expression> ">"  <expression>
            | <expression> ">=" <expression>
<expression>    = <term> <expressionpri>
                | "+" <term> <expressionpri>
                | "-" <term> <expressionpri>
<expressionpri> = "+" <term> <expressionpri>
                | "-" <term> <expressionpri>
                | ""
<term>    = <factor> <termpri>
<termpri> = "*" <factor> <termpri>
          | "/" <factor> <termpri>
          | ""
<factor> = <ident>
         | <number>
         | "(" <expression> ")"
         | "+" "(" <expression> ")" // extra
         | "-" "(" <expression> ")" // extra
         | "+" <number> // extra
         | "-" <number> // extra
         | "+" <ident>  // extra
         | "-" <ident>  // extra
<ident>  = `IDENT`
<number> = `NUMBER`
```

技术难点与实现方法

如何处理带符号表达式是这部分的主要难点，诸如 `-114` 或者 `+514` 之类的数字，又或者说是 `-homo` 这类的变量，再或者说是 `+(1919/810)` 这类的表达式前的符号。可以考虑将它们都归结到 `<factor>` 中实现，即增加前缀符号识别。

解析抽象语法树

这里使用的是**递归下降子程序法**解析抽象语法树，大体思路就是，通过递归函数栈来模拟匹配文法。

技术难点与实现方法

通过构造抽象转移图的方式存储CFG格式，进而模拟进行递归下降子程序法进行解析。

通过宏定义，将中间节点和终止节点的使用变得容易；Node通过返回自身地址，实现连续操作。

```
#define T(val) (parser.getTerminalNode(val))      // Terminal
#define N(name) (parser.getNonterminalNode(name)) // Nonterminal
N(statement) -> add({ N(ident), T(":="), N(expression) })
    -> add({ T("call"), N(ident) })
    -> add({ T("?"), N(ident) })
    -> add({ T("!"), N(expression) })
    -> add({ T("begin"), N(statement), N(statementpri), T("end") })
    -> add({ T("if"), N(condition), T("then"), N(statement) })
    -> add({ T("while"), N(condition), T("do"), N(statement) });
```

设计汇编原型与编译模式

首先设计基础汇编指令，一共分为 16 个基础指令，依次表示数据入栈、将栈顶数据弹出到变量、条件跳转、四则运算、条件判断（`odd` 表示判断是否是奇数，`#` 表示不等于）、控制台输入（`?`）和控制台输出（`!`）。

```
push, pop, jff, +, -, *, /, odd, =, #, <, <=, >, >=, ?, !
```

`push x` 会将立即数 x 压入栈顶，`push @x` 会将变量 `@x` 的值压入栈顶。

`pop @x` 会将栈顶元素弹出，并赋值给变量 `@x`。

`jff label` 会先弹出栈顶元素，如果栈顶元素为 0，则跳转到位置 `label`。

`+, -, *, /, =, #, <, <=, >, >=` 会弹出栈顶两个元素 x_1, x_2，然后将 $x_1 \oplus x_2$ 压入栈顶。

`odd` 会弹出栈顶元素 x，如果 x 是奇数，则压入 1，否则压入 0。

`?` 会从控制台读入一个整数（int类型），并压入栈顶。

`!` 会弹出栈顶元素并将其输出。

技术难点与实现方法

以下是详细的编译模式。

```
"const" <ident> "=" <number> <constdefpri> ";"  {  }  // 这部分在编译部分直接进行变量替换
"var" <ident> <vardefpri> ";" { push @@__stack_top; push <vert_ident>; +; pop @@__stack_top; } // 间接填充
```

```
"procedure" <ident> ";" <constdef> <vardef> <statement> ";" <procdef>
{
    push 0;
    jff nxt;
    &<ident>:;
    push @@__stack_top;
    push @@__stack_bottom;
    push @@__stack_top;
    push 1;
    +;
    pop @@__stack_bottom;
    [<constdef>]
    <vardef>
    <statement>
    push @@__stack_bottom;
    push 1;
    -;
    pop @@__stack_top;
    pop @@__stack_bottom;
    pop @@__stack_top;
    pop @@__ptr;
    nxt:;
}
<ident> ":=" <expression>  { <expression>; pop @<ident>; }
"call" <ident>    { push @@__ptr; push 4; +; push 0; jff &<ident>; }
"?" <ident>       { ?; pop @<ident>; }  // ?读入到栈顶,复用pop
"!" <expression>  { <expression>; !; }  // !输出栈顶并弹出,复用<expression>
"begin" <statement> <statementpri> "end"  { <statement> }
"if" <condition> "then" <statement>       { <condition>; jff nxt; <statement>;
nxt:; } // 如果<condition>为false则跳转(jump if false)
"while" <condition> "do" <statement>      { beg:; <condition>; jff nxt;
<statement>; push 0; jff beg; nxt:; }
"odd" <expression>                { <expression>; odd; }
<expression> "="  <expression>    { <expression1>; <expression2>; =; }
<expression> "#"  <expression>    { <expression1>; <expression2>; #; }
<expression> "<"  <expression>    { <expression1>; <expression2>; <; }
<expression> "<=" <expression>    { <expression1>; <expression2>; <=; }
<expression> ">"  <expression>    { <expression1>; <expression2>; >; }
<expression> ">=" <expression>    { <expression1>; <expression2>; >=; }
"+" <term> <expressionpri>    { <factor1>; <factor2>; +; }  // 弹出栈顶f1,f2,之后计
算f1+f2并放入栈顶
"-" <term> <expressionpri>    { <factor1>; <factor2>; -; }  // 弹出栈顶f1,f2,之后计
算f1-f2并放入栈顶
"*" <factor> <termpri>  { <factor1>; <factor2>; *; }  // 弹出栈顶f1,f2,之后计算
f1*f2并放入栈顶
"/" <factor> <termpri>  { <factor1>; <factor2>; /; }  // 弹出栈顶f1,f2,之后计算
f1/f2并放入栈顶
<ident>   { push @<ident> }
<number>  { push <number> }
"(" <expression> ")"  { <expression> } // <expression>在计算结束后会把值放入栈顶
"+" "(" <expression> ")" { <expression> }
"-" "(" <expression> ")" { push 0; <expression>; -; }
"+" <number> { push <number>; }
"-" <number> { push 0; push <number>; -; }
"+" <ident>  { push @<ident>; }
"-" <ident>  { push 0; push @<ident>; -; }
```

字节码的处理与执行

首先通过上述编译模式进行逐行翻译，获得汇编码；同样方式可以获得字节码。在编译完成后，通过虚拟机进行执行（这里用单栈机实现）。

技术难点与实现方法

在编译字节码时，会出现跳转地址不能立刻确定的情况，比如说 `<condition> jff nxt; <statement> nxt:` 这里的 `nxt` 地址不能在前面的 `jff nxt` 处确定。此时可以先记录 `jff nxt` 的语句地址，在填写完 `<statement>` 后，再去回填 `nxt` 地址。

```
if(nd -> astchild[0] -> val == "if") {
    // if <condition> do <statement>
    auto nxt = 0; // need to modify
    dfs(nd -> astchild[1]);
    auto idx = opcode.size();
    opcode.add(Opcode :: bc_jff(nxt)); // need to modify
    dfs(nd -> astchild[3]);
    opcode.modify(idx, Opcode :: bc_jff(opcode.size())); // here, rewrite `nxt`
address
}
```

参考资料

[1] wikipedia-Regular language

[2] wikipedia-Nondeterministic finite automaton

[3] wikipedia-Deterministic finite automaton

[4] wikipedia-Context-free grammar

[5] wikipedia-Abstract syntax tree

[6] wikipedia-LL parser

[7] wikipedia-Recursive descent parser

[8] wikipedia-Powerset construction

[9] wikipedia-Breadth-first search

[10] wikipedia-DFA minimization

[11] wikipedia-Left recursion