東南大學

编译原理课程设计

设计报告

组长：09017227 卓　旭

成员：09017224 高钰铭

09017225 沈汉唐

东南大学计算机科学与工程学院

二0 20 年 5 月

设计任务名称		SeuLex		
完成时间		2020-05-24	验收时间	2020-06-07
本组成员情况				
学　号	姓　名	承　担　的　任　务		成 绩
09017227	卓　旭	.l 文件的解析；正则表达式转 NFA；自动机可视化；词法分析器代码生成		
09017224	高钰铭	.l 文件的解析；NFA 转 DFA；词法分析器代码生成；C 字符串操作库		
09017225	沈汉唐	NFA 的合并；DFA 的最小化；C 代码美化功能		

1 编译对象与编译功能

1.1 编译对象

基本可以实现 C 语言全集作为编译对象。在 c99.l 中删除了少许不常用的词法定义（如 f 后标、LL 后标、<:符号），重写了一些不够严谨的正则表达式（如将字符串字面量重写为\"[^"\n]*\"），其余部分基本没有删改。

具体请参考 example/Simplified/c99_test.l。

1.2 编译功能

项目整体功能包括：

- .l 文件的解析，主要在 LexParser.ts 中完成；
- 正则表达式向加点、后缀形式的转换，主要在 Regex.ts 中完成；
- 由正则表达式构造 NFA，主要在 NFA.ts 中完成；
- 由 NFA 构造 DFA、DFA 最小化，主要在 DFA.ts 中完成；
- 词法分析器 C 代码的生成，主要在 CodeGenerator.ts 中完成。

项目还有一些特色功能，如自动机的可视化、C 代码的美化、C 代码的自动编译等。

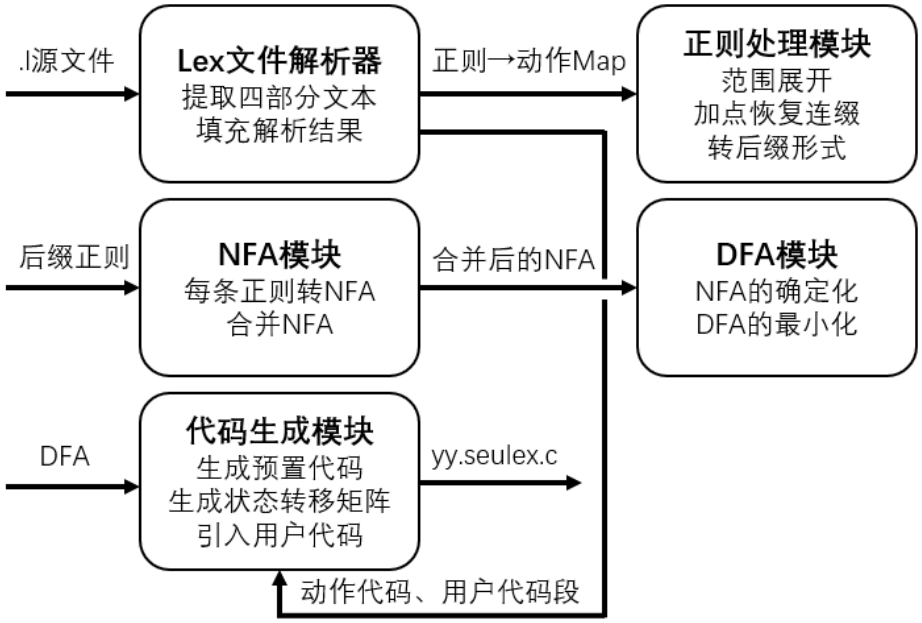
2. 主要特色

- 广泛使用面向对象编程，使代码结构清晰，各部分独立，之间易于相互调用
- 完整支持正则表达式五大元符号：? + * | .（0 或 1 次，1 次或以上，0 次或以上，或，任意字符）
- 正则定义部分可以使用或符|，让多条正则共享动作代码
- 支持正则表达式范围与范围补（ASCII 为全集）：[A-Za-z0-9_]、[^"]
- 支持正则别名定义
- 可以使用范围型转义字符\d（[0-9]）和\s（[\t\r\n]）
- 可在用户程序段重定向 yyout 输出流，默认为 stdout
- 支持 yylineno 获取行号、yyleng 获取词法单元长度、yytext 获取当前匹配字符串
- 支持 yyless 回退词法单元、yymore 补进词法单元
- 支持 yywrap 多输入文件连续处理、ECHO 输出 yytext
- 实现了最长匹配原则
- 正则表达式加点采用“隐式”加点处理，不向正则中插入任何字符，避免冲突
- 正则表达式中的“任意字符”（.）在自动机中不会展开为字符全集，而是采用巧妙的 ANY 边和 OTHER 边来表示，极大简化了状态
- 提供了增强功能：自动机可视化、自动调用 GCC 编译生成的 C 代码、美化最终生成的 C 代码、C 的字符串操作库
- 大量使用 assert 断言，程序具备一定的错误处理能力

3 概要设计与详细设计

3.1 概要设计

SeuLex 分为 Lex 文件解析器、正则表达式处理模块、NFA 模块、DFA 模块、代码生成模块、可视化模块。模块之间的关系和职责可以用如下的工作流图来表示：



3.2 详细设计

LexParser.ts 中的详细设计

Lex 文件解析器首先需要解析出.l 文件中的四部分：开头的直接复制部分、主体的正则-动作部分、末尾的用户 C 代码部分、夹杂在中间的正则别名部分，这可以通过对文件内容逐行扫描，通过定界符%%和%}等进行判别。注意解析过程中顺便判断.l 文件的结构是否符合要求，如果结构不正确需要报错处理。

接下来，需要对正则-动作部分进行进一步解析，提取出正则和相应的动作代码，并作别名展开。为了写出健壮的程序，支持千奇百怪的正则-别名部分的定义形式（单独一个分号表示没有动作、单句动作可以不加大括号、大括号可以换行也可以不换行、正则间可以使用或运算符，等等），我们使用了大量的状态变量来记录当前的状态，实现可靠的解析。这相当于手工构造了一个状态机。这本来可以通过 JavaScript 的正则表达式功能很简单可靠地实现，但在一个处理正则表达式的程序里使用正则表达式，总有投机取巧之嫌。我们也调查了 Flex 的做法，它的实现是写了一个描述.l 文件的.l 文件，然后使用 Lex 生成的词法分析器去分析.l 文件，这也是一种取巧的方法。

使用的状态记录变量

isReadingRegex = true, // 是否正在读取正则
isWaitingOr = false, // 是否正在等待正则间的“或”运算符
isInQuote = false, // 是否在引号内
isSlash = false, // 是否转义
isInBrackets = false, // 是否在方括号内
braceLevel = 0, // 读取动作时处于第几层花括号内

Regex.ts 中的详细设计

获得 LexParser 送来的正则表达式后，需要对正则表达式做一系列处理。首先要展开范围型转义字符\d（[0-9]）和\s（[\t\r\n]），这里有一些需要注意的点：①不能见到反斜杠就作转义，例如\\s 就不是对 s 的转义；②在非转义引号间的内容不能转义。

然后，展开方框范围。同样有如下需要注意的：①在非转义引号间的内容不处理；②要检查是否有方框重叠的情况;③展开 X-Y 时要检查 Y 的 ASCII 是否大于等于 X 的，否则要报错；④不要忘记处理剩余字符，如[0-9_\t]中还有一个下划线和 tab。在完成展开后，就可以用或符号|连接所有字符了。如果这是一个“取反型方框范围”（[^…]），则在字符全集上取补集即可。SeuLex 定义的字符全集是 ASCII 中的全体可打印字符。

至此，我们获得了规范化的正则表达式。下面进行加点处理，恢复连缀关系。我们考虑到用任何一个实际字符表示加点动作，都存在冲突的可能性，使得程序不够健壮，因此我们使用特别的“隐式加点”方法，使用数组来表示连缀关系，彻底避免冲突：

正则表达式	加点结果
a(b|c)+d*e?f	['a', '(b|c)+', 'd*', 'e?', 'f']

加点时的规则如下：

①非转义引号与其之间的内容要作为整体，不做加点处理；

②当前字符为定义的转义字符则不加点；

③当前字符为最后一个字符则不加点；

④当前字符为未转义的操作符|(则不加点

⑤下一个字符是操作符或右括号则不加点

最后，我们需要将中缀正则表达式转换为后缀正则表达式，便于后续处理。我们使用一个栈进行辅助。①对于引号内的内容，进行保持原样的打散，必要时添加转义，然后去掉引号；②遇到或符|时，优先级更低的是.*，全部弹出后加入自身；③遇到加点关系时，优先级更低的是.，全部弹出后加入自身；④遇到*+?时，直接加入自身；⑤遇到左括号(时，入栈；⑥遇到右括号)时，一直弹到左括号为止；⑦注意处理转义字符；⑧执行完成后栈内剩余直接追加到结果末尾。

FA.ts 中的详细设计

该文件主要定义了 NFA 和 DFA 的基类，以及其中用到的一些类型。

①自动机状态类 State。每一个状态使用一个全局唯一的 symbol（JavaScript ES6 标准的一种新数据类型，可作为 uuid）进行标识；

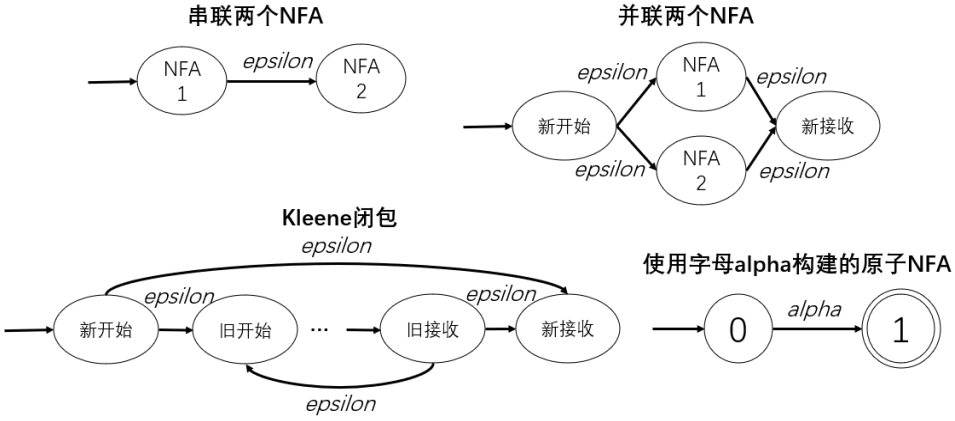
②自动机迁移边类 Transform。包括 alpha 属性（边上的字母在自动机字母表中的下标，使用-1 标识 epsilon，-2 表示 ANY，-3 表示 OTHER）和 target 属性（转移到的状态在自动机状态表中的下标）。

③有限状态自动机类 FA。包括字母表 alphabet（string 数组，使用字母表可以利用下标而不是字母本身定位字母，使用负下标表示特殊字母，这样功能强大且不易出错），状态数组 states，初始状态数组 startStates（通常只有一个），接收状态数组 acceptStates。另外，注意到自动机的状态数将会是巨大的，而连接边的数量是稀疏的，所以我们选用邻接链表来表示边，相比邻接矩阵可以节省大量空间，提高算法效率。

NFA.ts 中的详细设计

非确定有限状态自动机 NFA 类在继承 FA 类的基础上，添加了 acceptActionMap 属性，记录接收状态和动作代码的对应关系。

该类最关键的函数就是从后缀正则表达式构建 NFA 的 fromRegex 函数。首先，将后缀正则表达式拆解成单个字符，然后构造一个栈进行辅助。对于每一个字符：①如果是转义反斜杠，则进入转义字符状态，下一个字符成为被转义字符；②如果是或符|，则弹出栈顶两个 NFA，进行并联后放回；③如果是加点连缀符，则弹出栈顶两个 NFA，进行串联后放回；④如果是星闭包符*，则弹出栈顶一个 NFA，作 Kleene 闭包后放回；⑤如果是正闭包符+，则利用 A+是 AA*，即串联 A 和 A 作 Kleene 闭包后的结果后放回；⑥如果是零或一次符?，则弹出栈顶一个 NFA，在开始和接收态之间添加 epsilon 边后放回；⑦如果是任意字符点.，我们不需要把它展开成字符全集（否则会多出大量的状态，严重影响算法性能），而是引入了一个特别的 ANY 边，入栈一个 ANY 边的原子 NFA；⑧其他情况是普通字符的，入栈一个该字符为边的原子 NFA。一些示意图如下：



按如上规则构造后，最终栈内只剩下一个 NFA，即表示该正则表达式的 NFA。此时为该 NFA 的接收态绑定该正则表达式的动作代码即可。

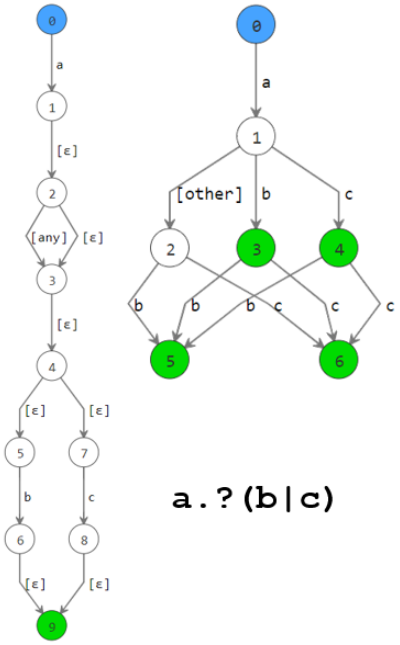
按照如上的方法，可以为 LexParser 中的所有正则表达式构造 NFA，最后并联成一个大的 NFA（但不添加新接收状态，而是使用原有的那些接收状态，否则动作代码无法绑定），即为该.l 文件定义的语言的 NFA 了。

DFA.ts 中的详细设计

确定有限状态自动机类构造时，接收一个 NFA，进行确定化和可选的最小化。

使用子集构造法从 NFA 构造 DFA。首先定义状态集合 I 的 epsilonClosure(I)，为 I 中的任何状态 S 经任意条 epsilon 边能到达的状态的集合；定义状态集合 I 的 a 弧转换 move(I, a)，表示可从 I 中的某一状态经过一条 a 弧到达的状态的全体。则 NFA 的确定化过程为：一开始，epsilonClosure(s0)（s0 是 NFA 的开始状态）是 Dstates（DFA 状态集合）中的唯一状态，且不加标记。当 Dstates 中仍存在未标记状态 T 时，标记它，然后对于每个输入符号 a，计算 U=epsilonClosure(move(T, a))，设置 Dtran[T, a]（DFA 的状态转移）为 U。并且，若 U 不在 Dstates 中，则将 U 不加标记地加入 Dstates。最后，子集中若包含 NFA 的接收态，则该 DFA 状态也是一个接收态。这一过程的具体算法可见龙书算法 3.20，此处不赘述细节。

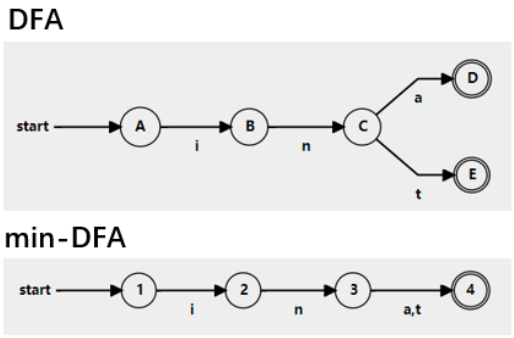
由于我们使用 ANY 边来表示表示任意字符的点号，因此在确定化过程中也需要特别处理。在确定化过程中，我们将 ANY 边视为可以接收任意字符的边进行子集构造。在确定化的最后，若某个 DFA 状态的出边同时包含 ANY 边和非 ANY 边，则将 ANY 边变成 OTHER 边以表示收到其他字符时的迁移。下面的示意图可以表示 ANY 和 OTHER 的工作机理（左侧为 NFA，右上为 DFA，右下为正则）：



另外，我们知道，子集构造法的每个 DFA 状态都对应于若干个 NFA 状态，那么在一个 DFA 接收态中，有没有可能包括多种动作代码不同的 NFA 接收态？我们发现，答案是肯定的。这种动作代码的冲突问题需要借助动作的优先级，即 Lex 的正则-动作部分“先定义的优先高”的规则进行解决。

在 DFA 的最小化方面，首先将状态划分为接收态和非接收态，然后对于每一个划分，如果存在状态 S 可以迁移到另一个划分，就把状态 S 移出该划分，以此类推，反复执行，直到划分不再变化，每个划分构成一个新的状态，即完成了 DFA 的最小化。需要注意的是，对于接收态，不能把它们放在同一个划分，而要在一开始就全部打散到独立的划分（即一个新状态不能包含多个接收态），这是因为接受态上还绑定着“动作代码”属性，因此它们之间不是真正等价的。

下面这张图说明了为什么 DFA 最小化时必须保留所有原始接收态。假设在做 C 语言的词法分析，int 和 ina 显然应采取不一样的动作。但如果合并了，动作将无法确定。



CodeGenerator.ts 中的详细设计

在完成了上述所有步骤后，即可生成词法分析程序的 C 代码。具体分为如下几步：

①拷贝 LexParser 解析出的直接复制部分；

②生成 seulex 的一系列预置变量，包括 yylineno、yyleng、yyin、yyout、yytext、yywrap 声明、ECHO 宏定义，以及其他一些应用 DFA 进行状态转移时必须的记录变量，如当前状态、最近接收状态、初始状态，等等；

③根据 DFA 生成状态转移矩阵_trans_mat，元素[i][k]表示在 i 状态收到 k 字符（ASCII 码）后转移到的状态的编号，-1 表示没有此转移。注意由于 DFA 中可能存在特殊的 ANY 边和 OTHER 边，因此每轮在设置完已知字母的转移后，要处理 ANY（全体字母）和 OTHER（剩余字母），将它们在矩阵的对应位置也设置值；

④生成接收状态到动作代码的 SwitchCase 的映射表，_swi_case[j]=x 在 x 大于等于 0 时表示是状态 j 是接收态，x 是动作代码的 case 编号；

⑤生成 yylex 函数，这是一个及其关键的函数，每当用户调用 yylex 时，seulex 就匹配一个新的词法单元，并执行对应的动作代码。

yylex 在匹配词法单元时，要遵循最长匹配原则。因此，每当到了接收状态，则暂时记录下来，然后继续进行状态转移，尝试接触下一个接收状态，进行更长的匹配。这样，记录的永远是当前看到的最长的接收状态；同长度情况下永远是最早出现接收状态（后出现规定不覆盖），符合 Lex 的优先级要求。直到无法进行更长的匹配，则把失败的多余匹配全部回退，然后采取最近记录的接收状态进行接收，填充 yytext 和 yyleng。

在 yylex 的过程中，碰到换行则 yylineno 加 1，碰到 EOF 后，如果 yywrap()不返回 1，则重置 yyin 指针及各个变量，开始新一轮词法分析。

⑥生成 yyless、yymore 函数，yyless(int n)允许用户回退 n 字符，yymore()允许用户追加移进一个词法单元。

⑦最后加上用户代码段部分，即完成了词法分析器代码生成。

增强功能的详细设计

①C 代码美化：生成的 C 代码格式可能不够美观，我们使用一些简单的方法，如 trim、分析大括号层数等，对代码进行了一定的格式化。下面是一个运行示例：

美化前	美化后

`int main() {`
`if (true) { prinf("hello"); }`
`}`
	`int main() {`
	`    if (true) {`
	`        prinf("hello");`
	`    }`
	`}`

②自动机可视化工具：在调试各算法的过程中，如果自动机只是看不见摸不着的一堆数据结构，对我们的调试效率有很大影响。因此，我们使用 dagre-d3 库，通过指定各节点显示内容、边上标签、背景颜色和迁移关系等，实现了自动机的可视化，前文“表示 ANY 和 OTHER 的工作机理的示意图”就是可视化功能的结果。这给我们的调试过程带来了极大的便利。

③C 字符串增强库：C 语言自身的字符串操作能力难以使用，为了方便用户在.l 文

件中对 yytext 等字符串进行操作，我们提供了一个字符串库，实现了包括取子串、trim、批量替换等一系列常用功能。

④GCC 自动调用：在调用 seulex 时，附加-c 参数，则 seulex 会使用 ChildProcess 自动唤起 GCC，传递合适的参数，编译生成的 yy.seulex.c 文件。

4 使用说明

SeuLex 使用说明

①从项目 GitHub 仓库（https://github.com/z0gSh1u/seu-lex-yacc/）下载或克隆整个仓库。

② seulex 依赖于 Node.js 运行时，因此你需要预先安装 Node.js（https://nodejs.org/zh-cn/）。然后在项目根目录下执行 npm install 来安装依赖。

③现在便可以使用下列命令运行 seulex 的 CLI 工具了：

node <path_to_project>/dist/seulex/cli.js <lex_file> <options>

<lex_file>为.l 文件的位置，我们为 C 语言准备的版本在 example/Simplified 目录下。

④可用的 options 包括：

-p：格式化生成的 C 代码

-c：生成后调用 GCC 编译，如果要给 GCC 传参，可使用"-c <params>"

-v：可视化最终的 DFA

⑤执行上述命令后，稍作等待，就会在当前目录下生成 yy.seulex.c 文件，即为词法分析器的 C 源代码，如图所示：

```
PS F:\seulexyacc> node ./dist/seulex/cli.js .\example\Simplified\c99_test.l -c
[ Running... ]
[ Parsing .l file... ]
[ Building NFA... ]
[ Building DFA... ]
[ Generating code... ]
[ Main work done! Start post-processing... ]
[GCC.JS] Calling gcc...
[GCC.JS] gcc called.
[ All work done! ]
[ Time consumed: 19458 ms. ]
```

⑥可以使用编译后的词法分析器分析 example/Snippet 下的 C 代码作为测试

⑦.l 文件有一些轻松的格式要求：

- 不可省略{% … %}部分
- 必须要在用户程序段为 yyin 输入流赋值为待分析文件流，这与原版 Lex 的行为是一致的
- 尽量不使用下划线开头的变量，因为可能与 seulex 预置变量冲突
- 如无特别情况，要定义一个 int yywrap()函数，始终返回 1，这与原版 Lex 的行为是一致的
- 正则表达式发展到今天，已经非常强大；Lex 本身也是一个强大的工具。限于自身水平，我们诚恳地承认，以下特性没有被支持：
  - 正则表达式不支持 {n, m} 等形式的出现次数限定
  - 不支持前瞻、后瞻正则表达式
  - 不支持 REJECT

5 测试用例与结果分析

使用 c99_test.l 生成的词法分析器进行实验。c99_test.l 的动作代码部分就是原样输出 TokenName 和对应的 yytext。结果如下：

测试用例 1（c99_test_1.c）

源代码:

```
int main() {
  float ad = 1.6;
  register long b = 1024;
  if (ad > b) {
    short i = 2;
    while (i -= 1) printf("hello world");
  } else {
    do {
      happy("tql?!");
    } while (1);
    b = 2 / 3;
  }
  return 0;
}
// end of file
```

分析结果（按列从上到下阅读），经验证完全正确:

TokenName	yytext	TokenName	yytext
INT	int	IDENTIFIER	printf
IDENTIFIER	main	LPAREN	(
LPAREN	(	STRING_LITERAL	"hello world"
RPAREN	)	RPAREN	)
LBRACE	{	SEMICOLON	;
FLOAT	float	RBRACE	}
IDENTIFIER	ad	ELSE	else
ASSIGN	=	LBRACE	{
CONSTANT	1.6	DO	do
SEMICOLON	;	LBRACE	{
REGISTER	register	IDENTIFIER	happy
LONG	long	LPAREN	(
IDENTIFIER	b	STRING_LITERAL	"tql?!"
ASSIGN	=	RPAREN	)
CONSTANT	1024	SEMICOLON	;
SEMICOLON	;	RBRACE	}
IF	if	WHILE	while
LPAREN	(	LPAREN	(
IDENTIFIER	ad	CONSTANT	1
GT_OP	>	RPAREN	)

IDENTIFIER	b	SEMICOLON	;
RPAREN	)	IDENTIFIER	b
LBRACE	{	ASSIGN	=
SHORT	short	CONSTANT	2
IDENTIFIER	i	SLASH	/
ASSIGN	=	CONSTANT	3
CONSTANT	2	SEMICOLON	;
SEMICOLON	;	RBRACE	}
WHILE	while	RETURN	return
LPAREN	(	CONSTANT	0
IDENTIFIER	i	SEMICOLON	;
SUB_ASSIGN	-=	RBRACE	}
CONSTANT	1	COMMENT	// end of file
RPAREN	)	——	

测试用例 2（c99_test_2.c）

源代码:

```
int Partition(int array[], int low, int high){
  int base = array[low];
  while(low < high){
    while(low < high && array[high] >= base){
      high--;
    }
    swap(array,low,high); // array[low] = array[high];
    while(low < high && array[low] <= base){
      low++;
    }
    swap(array,low,high); // array[high] = array[low];
  }
  array[low] = base;
  return low;
}
void QuickSort(int array[], int low, int high){
  if(low < high){
    int base = Partition(array,low,high);
    QuickSort(array,low,base - 1);
    QuickSort(array, base + 1, high);
  }
}
```

分析结果（按列从上到下阅读），经验证完全正确:

TokenName	**yytext**	**TokenName**	**yytext**
INT	int	IDENTIFIER	array
IDENTIFIER	Partition	COMMA	,
LPAREN	(	IDENTIFIER	low

INT	int	COMMA	,
IDENTIFIER	array	IDENTIFIER	high
LBRACKET	[	RPAREN	)
RBRACKET	]	SEMICOLON	;
COMMA	,	COMMENT	// array[high] = array[low];
INT	int	RBRACE	}
IDENTIFIER	low	IDENTIFIER	array
COMMA	,	LBRACKET	[
INT	int	IDENTIFIER	low
IDENTIFIER	high	RBRACKET	]
RPAREN	)	ASSIGN	=
LBRACE	{	IDENTIFIER	base
INT	int	SEMICOLON	;
IDENTIFIER	base	RETURN	return
ASSIGN	=	IDENTIFIER	low
IDENTIFIER	array	SEMICOLON	;
LBRACKET	[	RBRACE	}
IDENTIFIER	low	VOID	void
RBRACKET	]	IDENTIFIER	QuickSort
SEMICOLON	;	LPAREN	(
WHILE	while	INT	int
LPAREN	(	IDENTIFIER	array
IDENTIFIER	low	LBRACKET	[
LT_OP	<	RBRACKET	]
IDENTIFIER	high	COMMA	,
RPAREN	)	INT	int
LBRACE	{	IDENTIFIER	low
WHILE	while	COMMA	,
LPAREN	(	INT	int
IDENTIFIER	low	IDENTIFIER	high
LT_OP	<	RPAREN	)
IDENTIFIER	high	LBRACE	{
AND_OP	&&	IF	if
IDENTIFIER	array	LPAREN	(
LBRACKET	[	IDENTIFIER	low
IDENTIFIER	high	LT_OP	<
RBRACKET	]	IDENTIFIER	high
GE_OP	>=	RPAREN	)
IDENTIFIER	base	LBRACE	{
RPAREN	)	INT	int
LBRACE	{	IDENTIFIER	base
IDENTIFIER	high	ASSIGN	=

DEC_OP	--	IDENTIFIER	Partition
SEMICOLON	;	LPAREN	(
RBRACE	}	IDENTIFIER	array
IDENTIFIER	swap	COMMA	,
LPAREN	(	IDENTIFIER	low
IDENTIFIER	array	COMMA	,
COMMA	,	IDENTIFIER	high
IDENTIFIER	low	RPAREN	)
COMMA	,	SEMICOLON	;
IDENTIFIER	high	IDENTIFIER	QuickSort
RPAREN	)	LPAREN	(
SEMICOLON	;	IDENTIFIER	array
COMMENT	// array[low] = array[high];	COMMA	,
WHILE	while	IDENTIFIER	low
LPAREN	(	COMMA	,
IDENTIFIER	low	IDENTIFIER	base
LT_OP	<	MINUS	-
IDENTIFIER	high	CONSTANT	1
AND_OP	&&	RPAREN	)
IDENTIFIER	array	SEMICOLON	;
LBRACKET	[	IDENTIFIER	QuickSort
IDENTIFIER	low	LPAREN	(
RBRACKET	]	IDENTIFIER	array
LE_OP	<=	COMMA	,
IDENTIFIER	base	IDENTIFIER	base
RPAREN	)	PLUS	+
LBRACE	{	CONSTANT	1
IDENTIFIER	low	COMMA	,
INC_OP	++	IDENTIFIER	high
SEMICOLON	;	RPAREN	)
RBRACE	}	SEMICOLON	;
IDENTIFIER	swap	RBRACE	}
LPAREN	(	RBRACE	}

6 课程设计总结（包括设计的总结和需要改进的内容）

09017227 卓旭

seulex 的设计过程给了之前学习的编译原理的词法分析实验一次综合的复习与实践机会，让我们对词法分析技术有了更深刻的理解。在设计 seulex 的过程中，我们充分利用了 TypeScript 的模块化和强类型语言优势，尽可能构建了职能划分清晰、类型定义科学的代码，并且利用 some、every、map、filter 和 lambda function 等特性，初尝了函数式编程的冰山一角。

seulex 仍存在注释不够完善的问题，并且尤其在正则表达式处理阶段，有着处理流程不够清晰、统一的问题，有待改进。

09017224 高钰铭

项目中的 lex 进行实现时使用了一定的模块化设计，将 lex 文件解析、正则式 Regex、NFA、DFA 与代码生成各自作为一个独立的模块，通过各接口进行相互之间的交互。这一设计结构极大地方便了具体功能的开发过程。本人在 lex 部分的开发中主要实现了 NFA 转化为 DFA 的功能以及对 lex 文件解析的部分功能，实现过程基本按照预先搭好的架构进行，因此较为顺利，切实感受到了一个好的架构对程序开发的重要性。

但 lex 部分仍然有待改进之处，其中最为明显的是运行速度，因为开发过程中并没有注重考虑运行效率，因此许多地方存在一定的优化问题有待改进。

09017225 沈汉唐

这次的 lex 难度颇高且工程量巨大，其中最难的非 DFA 最小化莫属。lex 的设计是一个整体性很强、且注重循序渐进的过程。后面的内容都是建立在前面代码的基础上，如 NFA 是建立在 FA 的基础之上。因此当后面的子类需要某个函数的时候，往往要回到父类或者父类的父类补充函数。项目也存在需改进之处，如果考虑到效率的话，项目中存在的多重 for 循环确实是一个需要优化的部分。如果测试用的正则表达式极其庞大复杂，则会影响到程序运行的效率。

7 教 师 评 语

签名：_____

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做 PPT 文件、本设计报告等一切可放入光盘的内容的光盘。