南京林業大學
NANJING FORESTRY UNIVERSITY

2022～2023 学年第 1 学期 编译原理

课程实验

词法分析器，LL(1)/算符优先/SLR(1)语法分析器

学　　院：信息科学技术学院

专　　业：计算机科学与技术

课程名称：编译原理

学　　号：200855528

学生姓名：周航

指导老师：薛联凤

二 O 二二 年 11 月

一、词法分析器

实验意义

词法分析器可以将源文件中的字符串识别成 Token 流，也就是将字符串识别成许多有意义的单词，每个 Token 都有自己的 lexval，这个 Token 流可以传递给语法分析器，在符合语法的情况下将 lexval 综合属性在语法树上层层传递最终实现想要的语法效果。接下来就开始实现一个简单的词法分析器。

实验小结

根据 lex 文件，对输入的源文件进行词法分析。

首先描述需要的 lex 文件格式 lex 文件每行由两部分组成，用冒号分隔，冒号前部表示识别出来的 Token 名，后部分两种情况，如果用引号引住，说明要识别整个字符串，如果没有引号则按照正则语言识别，用分号表示一行的结束，并且支持使用'//'用于单行注释，示例如下：

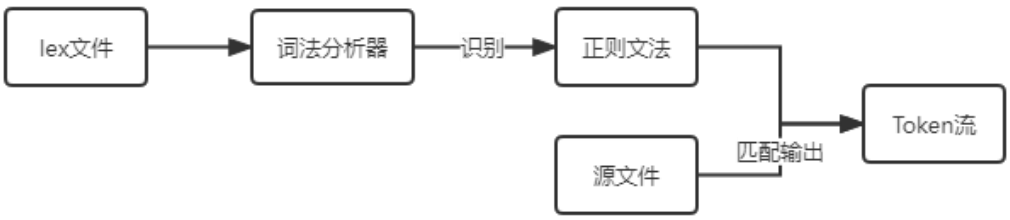
（1）、关键字这样描述 “Char : 'char';”表示将 char 字符串识别为一个整体，命名为 Char

（2）、对于符号，这样描述 “Plus : '+';”表示将加字符识别出来，命名为 Plus

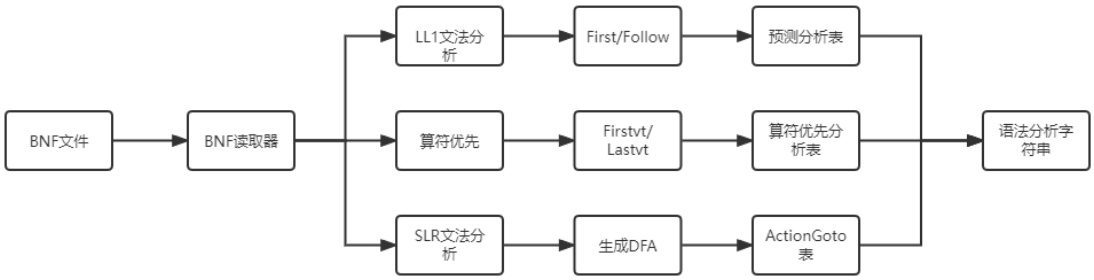
（3）、对于数字信息，这样描述:“DecimalConstant : [1-9][0-9]* ;”这是十进制正整数的正则语言描述，并命名为 DecimalConstant，支持任何正则文法。

获得 lex 文件后程序将其读取，对每一个 Token 生成对应的正则表达式。接着给定一个源文件，对这个文件的每一行，依次使用上述正则表达式去匹配，同时记录行号列号，最终将源文件拆分为 token 流。

对于错误处理，只需要在正则匹配没有匹配成功时提示相应错误即可。



二、语法分析器



2.1 BNF 读取器

实验意义

对于语法分析来说，BNF 是最重要的输入文件了，语法分析器应该遵循 BNF 文件的指导进行语法分析，所以有一个完善的 BNF 读取器对于后面的实验来说是很重要的，该读取器可以将 BNF 文件中的非终结符和终结符信息，产生式信息以容易读取的形式存储起来，便于后面读取。

实验小结

/src/BNF.py 包含了将原始BNF语法识别为Token流的抽象(又是一个小的词法分析!),算是封装的很成功的模块,在后面的所有算法中都用到了,能够读取 bnf 文件,支持的语法如下

(1)、 只支持单层的 | ,不支持括号嵌套语法

(2)、 每个 Token 之间用空格分隔,不支持尖括号指定 Token,所以不需要加字符串来指定非终结符

(3)、 认定 ::= 符号前面的都是非终结符,其他都是终结符,所以不限制非终结符一定要大写

(4)、支持单行使用 # 注释,不支持行内注释和多行注释

(5)、将 ε 字符视为 epsilon,其他均不可

一些符合语法的 BNF 文件示例放在 test/下,对于简单的例子,学习编译原理而言是足够使用的,下面展示一个符合要求的 BNF 文法文件。

```
1. E ::= E + T | T
2. T ::= T * P | P
3. # F ::= P ↑ F | P 这是注释
4. P ::= ( E ) | i
```

2.2 LL(1)分析器

实验意义

自顶向下分析中的经典算法之一,著名的 Antlr 就是采用的自顶向下分析技术,然而该技术需要手动先将 BNF 转化为非左递归的形式,这给编译器开发带来很大困难,因为大部分文法都或多或少包含左递归,所以该分析如果想要真正有实际作用先要开发一个能够自动消除左递归的程序。但是算法本身相较自底向上分析实现起来是容易很多的,计算量也比自底向上要少一些,所以还是值得深入研究的。

实验小结

/src/Up_Down.py 自顶向下分析算法中的 LL(1)分析器,可以依次生成 First 首符集/Follow 随符集/预测分析表,并在此基础上实现递归下降分析,识别一个句子是否符合输入的 BNF 语法,例如经典的表达式文法(已经手动消除左递归处理的 BNF)

```
1. E ⟹ T E'
2. E' ⟹ + T E' | ε
```

```
3. T ⟹ F T'
4. T' ⟹ * F T' | ε
5. F ⟹ ( E ) | i
```

可以生成如下的信息

	非终结符	首符集(FIRST)	随符集(FOLLOW)
1	E	i (	) #
2	E'	ε +	) #
3	T	i (	) # +
4	T'	* ε	) # +
5	F	i (	) * # +

		+	ε	*	(	)	i	#
1	E	Error	Error	Error	E ⟹ T E'	Error	E ⟹ T E'	Error
2	E'	E' ⟹ + T E'	Error	Error	Error	E' ⟹ ε	Error	E' ⟹ ε
3	T	Error	Error	Error	T ⟹ F T'	Error	T ⟹ F T'	Error
4	T'	T' ⟹ ε	Error	T' ⟹ * F T'	Error	T' ⟹ ε	Error	T' ⟹ ε
5	F	Error	Error	Error	F ⟹ (E)	Error	F ⟹ i	Error

	下推栈	当前剩余字符串	动作
1	[#, E', T]	i+i#	E ⟹ T E' 反序压栈
2	[#, E', T', F]	i+i#	T ⟹ F T' 反序压栈
3	[#, E', T', i]	i+i#	F ⟹ i 反序压栈
4	[#, E', T']	+i#	直接匹配
5	[#, E']	+i#	T' ⟹ ε 反序压栈
6	[#, E', T, +]	+i#	E' ⟹ + T E' 反序压栈
7	[#, E', T]	i#	直接匹配
8	[#, E', T', F]	i#	T ⟹ F T' 反序压栈
9	[#, E', T', i]	i#	F ⟹ i 反序压栈
10	[#, E', T']	#	直接匹配
11	[#, E']	#	T' ⟹ ε 反序压栈
12	[#]	#	E' ⟹ ε 反序压栈
13	[]		直接匹配

2.3 算符优先分析器

实验意义

算符优先分析器算是入门自底向上分析思想的最合适的算法，虽然这个算法有自身的局限性，只能识别很小的一部分算法。

实验小结

/src/Operator_First.py 算符优先文法分析器，可以依次生成 Firstvt/Lastvt/算符优先表并可视化分析过程，还是以经典的表达式文法为例（不需要左递归处理）

```
1. E ⟹ E + T | T
2. T ⟹ T * P | P
```

```
3. P ⟹ ( E ) | i
```

生成 Firstvt/Lastvt 表，✔表示该终结符属于 Firstvt 集合，星星表示该终结符属于 Lastvt 集合

		+	*	(	)	i
1	E	✓/⭐	✓/⭐	✓/	/⭐	✓/⭐
2	T	/	✓/⭐	✓/	/⭐	✓/⭐
3	P	/	/	✓/	/⭐	✓/⭐

		+	*	(	)	i	#
1	+	⩾	⩽	⩽	⩾	⩽	⩾
2	*	⩾	⩾	⩽	⩾	⩽	⩾
3	(	⩽	⩽	⩽	=	⩽	
4	)	⩾	⩾		⩾		⩾
5	i	⩾	⩾		⩾		⩾
6	#	⩽	⩽	⩽		⩽	=

根据算符优先表分析 i+i 串的过程

	分析栈	当前剩余字符串	动作
1	[#, i]	+i#	移进 i
2	[#, P]		规约 [i] → P
3	[#, P, +]	i#	移进 +
4	[#, P, +, i]	#	移进 i
5	[#, P, +, P]		规约 [i] → P
6	[#, E]		规约 [P, +, P] → E

算符优先是最小巧我最满意的，没有发现严重的 bug，缺点是能识别文法太少了，这也是算法本身的问题

2.4 SLR(1)分析器

实验意义

普通的 LR(0)分析算法很容易出现 ActionGoto 表项的冲突，能识别的文法也不多，SLR 分析器使用 Follow 集合作为前看符号，大大扩展了能够识别的文法范围，这其中为什么使用 Follow 集合就能消除很多冲突是值得研究的。

实验小结

/src/SLR.py 实现 SLR(1)分析器，是截止当前最复杂的，首先将产生式分解成带有活前缀的项目，据此生成识别活前缀的 DFA，再根据 DFA 生成 action_goto 表格，再根据该表格实现 SLR(1)分析。整个过程中，生成 DFA 的过程相比于正则引擎需要汤普森算法需要递归，这里只需要循环迭代即可生成，然而还是在理解上遇到困难，并且现在还是存在一些小问题，

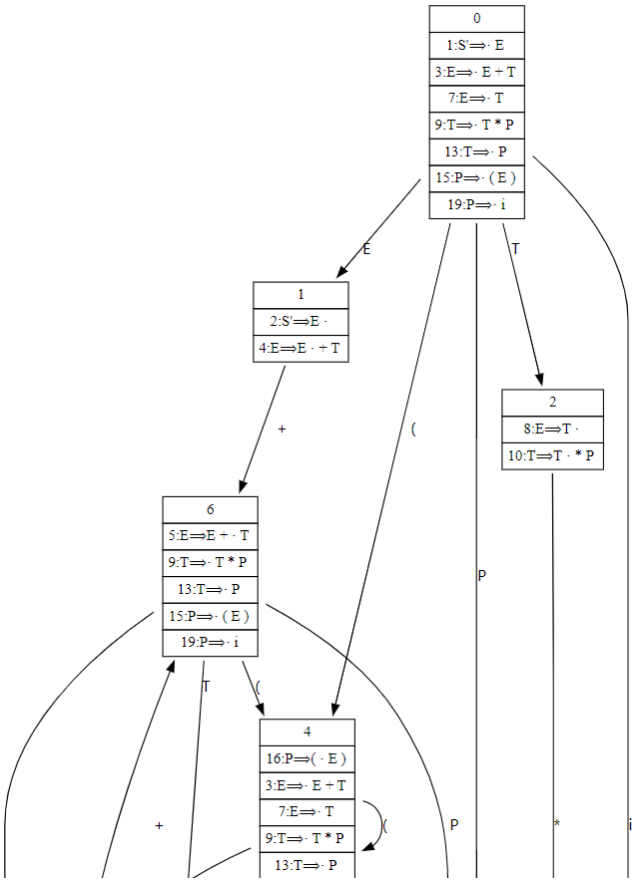
例如在生成新的 DFA 节点时，在某些情况下无法合并到已经存在的 DFA 节点上。依然以经典的表达式文法为例：

```
1. E ⟹ E + T | T
2. T ⟹ T * P | P
3. P ⟹ ( E ) | i
```

生成的 action_goto 表，总共四种表项状态，ACC/Error/规约/移进。

	+	*	(	)	i	#	E
0	Error	Error	Shift4	Error	Shift5	Error	1
1	Shift6	Error	Error	Error	Error	ACC	Error
2	8:E⟹T ·	Shift7	Error	8:E⟹T ·	Error	8:E⟹T ·	Error
3	14:T⟹P ·	14:T⟹P ·	Error	14:T⟹P ·	Error	14:T⟹P ·	Error
4	Error	Error	Shift4	Error	Shift5	Error	8
5	20:P⟹i ·	20:P⟹i ·	Error	20:P⟹i ·	Error	20:P⟹i ·	Error
6	Error	Error	Shift4	Error	Shift5	Error	Error
7	Error	Error	Shift4	Error	Shift5	Error	Error
8	Shift6	Error	Error	Shift12	Error	Error	Error
9	8:E⟹T ·	Shift7	Error	8:E⟹T ·	Error	8:E⟹T ·	Error
10	6:E⟹E + T ·	Shift7	Error	6:E⟹E + T ·	Error	6:E⟹E + T ·	Error
11	12:T⟹T * P ·	12:T⟹T * P ·	Error	12:T⟹T * P ·	Error	12:T⟹T * P ·	Error
12	18:P⟹(E) ·	18:P⟹(E) ·	Error	18:P⟹(E) ·	Error	18:P⟹(E) ·	Error

生成的 DFA（部分）



对于字符串 i+i 的分析过程展示

	状态栈	符号栈	当前剩余字符串	动作
1	[0]	['#']	i+i#	
2	[0, 5]	['#', 'i']	+i#	移入 i
3	[0, 3]	['#', 'P']	+i#	规约 20:P⟹i ·
4	[0, 2]	['#', 'T']	+i#	规约 14:T⟹P ·
5	[0, 1]	['#', 'E']	+i#	规约 8:E⟹T ·
6	[0, 1, 6]	['#', 'E', '+']	i#	移入 +
7	[0, 1, 6, 5]	['#', 'E', '+', 'i']	#	移入 i
8	[0, 1, 6, 3]	['#', 'E', '+', 'P']	#	规约 20:P⟹i ·
9	[0, 1, 6, 10]	['#', 'E', '+', 'T']	#	规约 14:T⟹P ·
10	[0, 1]	['#', 'E']	#	规约 6:E⟹E + T ·

三、总结

本次实验我使用 Python 实现，使用 PyQt 以表格形式可视化分析过程中使用到的数据结构，使用 graphviz 调用 dot 语言来对 DFA 图进行可视化，显示效果也是出乎我的预料，为了将其放在 qt 界面中展示，我将生成的 DFA 图转化为 svg，并给它一个不错的 html 框架显示，再使用 qt 的 QWebEngine 渲染 html，这样就将图片完美嵌入到图形界面中了。本次实验最大的感触就是理论和实际的区别还是很大的，光是理论学明白了并不代表真的理解某个算法，等到真正上手写程序了才会发现算法背后的含义，为什么设计成这个样子，这些算法并不是冰冷的数学公式，都是在程序的背景下发明出来的，适合计算机的算法。

不足之处是在编写 SLR 文法分析器的时候没能考虑到为 LR1 文法预留一个前看符号的空间，导致现在如果想在 SLR 的基础上升级成 LR1 文法，有大量需要修改的地方，整体的冗余非常多，还有在编写 SLR 文法时，一开始对于细节问题没有重视导致写了很多没用的代码，例如由于无法保证给定的 BNF 文件符合 SLR 文法，在生成 ActionGoto 表时可能会发生表项冲突的问题，此时最好的解决方法应该是报错提示退出，然而我花费很多时间为 ActionGoto 开发了冲突检测的功能，也就是同一个表项可以显示多个动作。

最后我觉得未来的路任重而道远，这门课仅仅是带领我入门了编译的世界，我也仅仅是实现了一些语法分析的算法，实现语法分析仅仅是实现编译器的第一步，后面还有更加困难复杂的工作，比如如何将词法分析得到的 token 流连接到语法分析器上进而做一些语义分析，生成中间代码的工作，再进一步，如何优化这个中间代码，最终根据需要生成对应 ISA 的可以运行在真实硬件上的汇编代码，这也是我日后的研究目标之一。

完整源码开源在此处 Zre: Dive Into Regex (gitee.com)