

编译原理实验二报告

09015113 柳乔丰

1. Motivation/Aim

Program a simple Syntax Analyzer.

2. Content description

Input: Stream of characters

- Assume that they are given in the file `“./Input.txt”`

Context-Free Grammar

- Assume that they are defined in the file `“./Language.txt”`
- Also, additional **priority** relationship are defined in the file `“./Priority.txt”`

Output: Sequence of reductions

- The result is displayed in the file `“./Analysis Result.txt”`

（因为时间有限，所以我在这里没有做 Error Handling。但是，代码包含了 Error Detection，体现在 `LR_Parser.hpp` 中的 `parseLine` 方法。）

3. Ideas/Method

我使用了 **Soft Coding** 的方式实现了 **Canonical LR(1) Parser**，整体构造流程如下：

1. 用户通过修改 `./Language.txt` 文件添加自己定义的语句（按照以下规范：**“Production \$(items in Production, seperated by space)\$”**）
2. 用户修改 `./Priority.txt` 文件更改终结符的优先级（优先级高在前）
3. 程序基于输入分别对每个产生式构造对应的 **Production** 对象。
4. 程序将这些对象最终合并并构造一个 **Language** 对象
5. 程序由 **Language** 对象构造 **LR_DFA**
6. 程序最终构造 **LR_Parser**，**LR_Parser** 中包含了构造的 **LR_DFA**
7. 程序调用 **LR_Parser** 类中的 `parseLine` 方法来对输入文件的每一行（也就是每个输入序列）进行语法分析
8. 程序对每个输入序列输出相应的分析结果（也就是规约序列）

4. Assumptions

基础假设如下：

1. 语句及优先级已经预先定义，其中对于每个产生式，产生式中的所有符号也预先给定。（都已定义在了 `./Language.txt` 文件中）
2. 定义的语句遵循一定的文法：必须是增广语法；可以有二义性；必须满足 Canonical LR(1) Parser 的使用条件。（在分析表中，最终不会出现“移入-移入”冲突和“规约-规约”冲突（在已经处理过二义性后））
3. 语句中不含有除“|”外的正则表达式运算符，正则表达式的模式已经通过匹配给出，语句定义符合一定的规范。
4. 编号为 0 的产生式的左部为增广语法语句的开始符，所有其它的产生式中都不包含增广文法语句的开始符。

5. Related FA descriptions

LR_DFA 是由语法分析程序生成的。(生成 LR_DFA 的代码部分为 LR_DFA.hpp 中的 **construct** 函数) 生成 LR_DFA 的思路参照课本描述。(详细的例子描述参照龙书 P167 Example 4.54) 因为 DFA 的构造过程大致类似, 所以我直接复用了大部分我自己实验一的代码。(对 DFA 的构造思路和之前类似, 都是先构造当前的状态; 然后通过当前状态, 得到各个出边 (对于非终结符的 GOTO 我在这里按照 SHIFT 动作处理), 然后预先构造好/查询到下一个状态的部分内容 (包括了下一个状态的核, 点的位置, 以及向前看符等); 最后, 递归构造下一个状态。) 同样, 因为按照 LALR(1) Parser 的构造方法合并状态可能会造成“规约-规约”冲突, 所以我并没有合并 DFA 中的状态。

6. Description of important Data Structure

- 1) **LR_DFA::Item**: Canonical LR(1)项, 类的各个字段定义如下:
 - **productionID**: 对应的产生式编号 (对每个产生式已预先编号)
 - **dot**: 当前产生式中, 点所在的位置 (比如 $A \rightarrow b.B$ 中, 点在第 1 位)
 - **lookAhead**: 向前看符

(我还重载了 “!=”、“==”、“<” 这几个运算符, 为了能够顺利使用 STL 中提供的 set 容器。这是因为 set 容器定义时必须给出容器中每个元素的大小关系。)

- 2) **LR_DFA::State**: LR_DFA 的状态表示类, 类的各个字段定义如下:
 - **core**: LR_DFA::Item 的集合, 即每个状态的核
 - **id**: 状态的编号
 - **edgeCount**: 出边的个数
 - **links**: 指针类型, 其中, **links[i].To** 为第 i 条边指向的 DFA 状态; **links[i].edge** 为第 i 条边上的符号, **links[i].action** 为第 i 条边转移到下一个状态的动作: 可以是 SHIFT (对非终结符 GOTO 也按照 SHIFT 处理), ACCEPT, 也可以是一个大于 0 的数 (表示动作是 REDUCE, 值为规约所使用的产生式的编号)。

3) **LR_DFA**: 基于 Language 对象构造的 DFA。LR_DFA.hpp 文件中, **construct** 方法用于构造 DFA; **closure** 方法用于计算闭包; **shift** 方法用于状态间的转移 (在当前状态下, 判断是否有出边上的符号与输入给定的符号相同, 如果有则转移到下一个状态。这里还未考虑有多条边上给定符号相同的情况, 留到后续 LR_Parser 中做处理, 也就是处理文法的二义性)。

- 4) **LR_Parser**: 最终的语法分析器。内部包含一个 LR_DFA。LR_Parser.hpp 文件中, **parseLine** 方法为用于语法分析; **displayTable** 方法为查看由 LR_DFA 构造的得到的 Parsing Table。

7. Description of core Algorithm

1. 计算 FIRST

FIRST 函数的实现算法在龙书 4.4.2 节中已经阐述过：

计算文法符号 X 的 $FIRST(X)$ 时：

- 1) 如果 X 是一个终结符，则 $FIRST(X) = X$ 。
- 2) 如果 X 是一个非终结符，且 $X \rightarrow ABC\dots$ 是一个产生式，则对产生式右部：如果 A 可以推出 ϵ ，则在将 $FIRST(A)$ 中所有的符号都加入到 $FIRST(X)$ 中的同时，还要将 A 之后的 B 中的 $FIRST(B)$ 加入到 $FIRST(X)$ 中，直到某个符号不能推出 ϵ 为止。如果产生式右部的所有符号都能推出 ϵ ，则最后将 ϵ 也加入到 $FIRST(X)$ 中。
- 3) 如果 $X \rightarrow \epsilon$ 是一个产生式，则将 ϵ 加入到 $FIRST(X)$ 中。

我对 FIRST 函数的实现思路也与上面类似，但是，因为可能会出现 infinite loop 的情况（对于左递归存在的时候），所以需要判断，避免嵌套计算的情况。（当然，也可以事先定义好一个不存在左递归的文法。）
(实现细节可以参见 Language.hpp 中的 first 方法。)

2. 计算闭包（自动机内部状态扩展）

计算闭包的算法已在龙书 P167 Algorithm 4.53 中给出，这里在细节部分，我做了一些修改。算法过程如下：

不断重复以下过程，直到集合 I 中不能再加入更多的项：

1. 对于每个 I 中的项，取出点之后的符号，判断是否为非终结符
2. 对于这些非终结符，找到由它构成左部的产生式。同时，找出当前点之后符号的后一个符号 β ，求出 $FIRST(\beta a)$ ，其中 a 为向前看符。（如果 β 为 ϵ ，则置 β 为空。）
3. 将 $[B \rightarrow \gamma, b]$ 加入到集合 I 中， b 为 $FIRST(\beta a)$ 集合的元素。
(实现细节可以参见 LR_DFA.hpp 中的 clousure 方法。)

3. LR_DFA 的构造算法

LR_DFA 的构造思想在第 5 点描述 FA 中已经给出。具体思路和上次实验构造 LR_DFA 类似，结合了龙书 P167 Algorithm 4.53 中的 GOTO 函数的实现算法。LR_DFA 的构造算法整体流程如下：

不断重复以下过程，直到所有构造完成：

1. 对于当前状态，取出状态的核中所有点之后的符号，加入到 after_dot 集合中；特别的，如果点在产生式右部的末尾处，则将 lookAhead 加入到 reduce 数组中，同时将要规约的产生式编号加入到 reduceID 数组中（reduce 中的符号和 reduceID 一一对应）。
2. 以下进入循环，对每个 after_dot 集合内的元素 A ：
 - 1) 在当前状态的核中寻找点之后符号和 A 一致的产生式。找到后，将产生式加入到新的核中，点的位置后移一位，向前看符 lookAhead 保持不变。
 - 2) 对步骤 1) 中构造的 LR 项计算闭包，完成一个 LR 项的构造。

- 3) 在现有的状态中查看这个 LR 项是否出现过（保存 LR 项的内容和 LR 项的地址这个一一对应关系，方便使用检索）：如果这个 LR 项没有出现过，则新建一条边、新建一个状态，边上的符号为 A，边的动作为 SHIFT，指向的下一个状态的 LR 项为步骤 2) 中构造的 LR 项，以上完成后，继续递归下一个状态；如果这个 LR 项出现过，则新建一条边，边上的符号为 A，边的动作为 SHIFT，指向的下一个状态为检索出的那个状态。
 3. 对于数组 `reduce` 中的每个元素 B，新建一条边、新建一个状态，边上的符号为 B，边的动作为 REDUCE，指向的下一个状态（这个状态中没有 LR 项，不将这个状态加入记录存在状态的数组中）。
- （实现细节可以参见 `LR_DFA.hpp` 中的 `construct` 方法。）

4. LR_Parser 的语法分析算法

语法分析算法的实现参照了中文龙书 P160 Figure 4-36 给出的 LR 语法分析伪代码。LR 语法分析程序的整体流程如下：

取 `a` 为输入序列的第一个符号，永远重复如下过程：

找到边上符号为 `a` 的动作 `action`（利用之后提及的处理文法二义性的方法【在本次报告第 9 点对问题的讨论和解决中提到】来解决可能出现的冲突）。当前动作 `action` 可能有以下四种情况：

1. 如果当前动作 `action` 为 SHIFT，将 `a` 压入符号栈（如果 `a` 为终结符，则同时将它压入终结符栈），利用 LR_DFA 的 `shift` 函数转移到这个动作对应边指向的状态，然后将这个状态压入状态栈。取 `a` 为下一个输入符号。
2. 如果当前动作 `action` 大于 0（也就是当前动作为 REDUCE），那么首先，将编号为 `action` 的产生式加入到输出结果 `ans_seq` 中。之后，从符号栈栈顶弹出 `b` 个符号（`b` 为编号为 `action` 的产生式右部符号的个数），终结符栈弹出这 `b` 个符号中的所有的终结符，同时从状态栈中弹出 `b` 个状态。把 `cur_state` 置为状态栈栈顶的状态。取 `A` 为规约表达式左部的符号。将 `A` 压入符号栈，利用 LR_DFA 的 `shift` 函数转移到这个动作指向的状态，将这个状态压入状态栈。
3. 如果当前动作 `action` 为 ACCEPT，则语法分析完成。
4. 如果当前动作 `action` 为其它，则输出 ERROR。

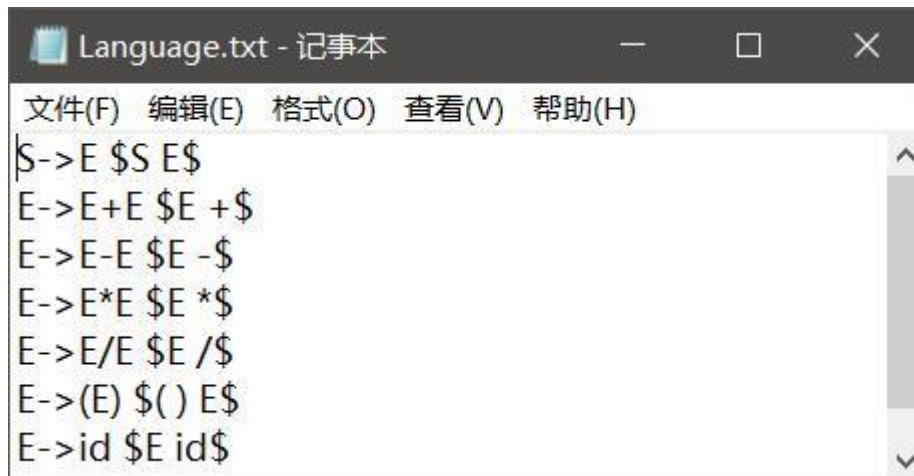
（实现细节可以参见 `LR_Parser.hpp` 中的 `parseLine` 方法。）

8. Use cases on running

(所有的测试文件均在/lab2/lab2 文件夹下，仅保留了测试 sample2)

测试 sample1:(二义性文法)

文法定义文件 (./Language.txt) :



```
Language.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
S->E $S E$
E->E+E $E +$
E->E-E $E -$
E->E*$E $E *$
E->E/E $E /$
E->(E) $( ) E$
E->id $E id$
```

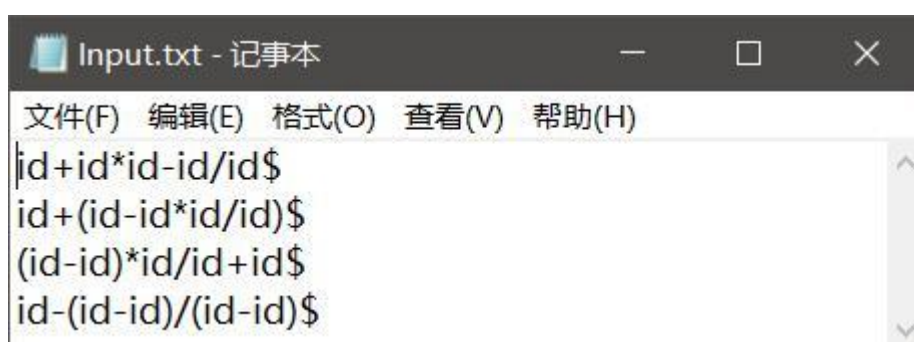
优先级定义文件 (./Priority.txt) : (优先级高的符号定义在前)

(这里定义的优先级是绝对优先级，也就是说不允许相同优先级存在)



```
Priority.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
(
)
*
/
+
-
```

输入文件 (./Input.txt):



```
Input.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
id+id*id-id/id$
id+(id-id*id/id)$
(id-id)*id/id+id$
id-(id-id)/(id-id)$
```

输出文件 (./Analysis Result.txt): (规约序列)

```
Analysis Result.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
===== Analysis Result 01 =====
E->id
E->id
E->id
E->E*E
E->E+E
E->id
E->id
E->E/E
E->E-E
=====
===== Analysis Result 02 =====
E->id
E->id
E->id
E->id
E->E*E
E->id
E->E/E
E->E-E
E->(E)
E->E+E
=====
```

```
Analysis Result.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
===== Analysis Result 03 =====
E->id
E->id
E->E-E
E->(E)
E->id
E->E*E
E->id
E->E/E
E->id
E->E+E
=====
===== Analysis Result 04 =====
E->id
E->id
E->id
E->E-E
E->(E)
E->id
E->id
E->E-E
E->(E)
E->E/E
E->E-E
=====
```


测试 sample2:(二义性文法，经典的悬空 else 问题)

文法定义文件 (./Language.txt) :

```
Language.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
B->S $B S$
S->ifSthenS $S if then$
S->ifSelseS $S if else$
S->ifSthenSelseS $S if then else$
S->{S} $S {$}$
S->A $S A$
```

优先级定义文件 (./Priority.txt) : (优先级高的符号定义在前)
(这里定义的优先级是绝对优先级，也就是说不允许相同优先级存在)

```
Priority.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
{
}
else
then
if
```

输入文件 (./Input.txt):

```
Input.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
ifAthenifAthenAelseA$
ifAthen{ifAthen{ifAthenAelseA}elseA}$
ifAthenifAthen{ifAthenAelseA}elseA$
```

输出文件 (./Analysis Result.txt): (规约序列)
(由./Priority.txt 文件中定义可以看出, 这里我假设 **else** 匹配最近的 **if**)

```
Analysis Result.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
===== Analysis Result 01 =====
S->A
S->A
S->A
S->A
S->ifSthenSelseS
S->ifSthenS
=====
===== Analysis Result 02 =====
S->A
S->A
S->A
S->A
S->A
S->ifSthenSelseS
S->{S}
S->A
S->ifSthenSelseS
S->{S}
S->ifSthenS
=====
```

```
Analysis Result.txt - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
S->A
S->A
S->ifSthenSelseS
S->{S}
S->A
S->ifSthenSelseS
S->{S}
S->ifSthenS
=====
===== Analysis Result 03 =====
S->A
S->A
S->A
S->A
S->A
S->ifSthenSelseS
S->{S}
S->A
S->ifSthenSelseS
S->ifSthenS
=====
```


9. Problems occurred and related solution

1. 如何解决文法的二义性问题？

LR(1) Parser 使用的时候可能会碰到具有二义性的文法，它的特点是构造出来的 Parsing Table 中可能会有“移入-规约”冲突。解决文法的二义性问题老师上课的时候也讲过，就是终结符的优先级。我选择不在构造 LR_DFA 的时候解决，而是在最后在 LR_Parser 的 parseLine 中处理这个问题。

对比龙书提供的思路，我额外新增符号栈和终结符栈。其中，终结符栈起到最关键的作用（符号栈是为了终结符栈服务的，当符号栈中一个终结符出栈时，终结符栈中也同时出栈一个终结符）。

二义性问题整体解决思路如下：

在语法分析的时候，因为我的算法是预先取出动作，在这个时候，对第一次取输入符号对应的动作时，不加处理；在第二次取动作的时候（一个符号对应多个动作，可知这个时候发生冲突了），将当前指标指向的输入符号 a 与终结符栈的栈顶符号 b 进行比较（如果终结符栈为空，那么可以跳出这个过程）。如果符号 a 对应动作为 SHIFT，且输入符号 a 的优先级高于终结符栈顶符号 b，那么置当前动作为 SHIFT；否则不做处理。

（这里我运用到了之前的假设：对于强大的 LR(1) Parser 来说，除了会碰到二义性问题之外，不会再有“移入-移入”冲突和“规约-规约”冲突。所以仅仅只需考虑解决“移入-规约”冲突即可。）

（实现细节可以参见 LR_Parser.hpp 中的 parseLine 方法。）

2. 如何检索到 DFA 中的某个状态？

前面提到的构建 DFA 的过程中，当前状态转移到的下一个状态可能是已经存在的状态，如何检索这个状态呢？（值得注意的是，在遍历状态检索的过程中可能会碰到环。）我的想法是利用一个 struct 同时储存核（core）和对应该状态的地址。那么在检索的时候，仅需利用一个指向指针的指针即可找到这个检索的地址。因为 STL 中容器不支持直接返回容器的地址，所以这属于无奈之举。（实现细节可以参见 LR_DFA.hpp 中的 indexCore 方法。）

3. 如何优化语法分析程序？

由龙书上介绍可知，我们在做语法分析的时候是表格驱动的，也就是说构造 LR_DFA 是为了构造 LR Parsing Table 而服务的，也就是说终极目标是构建出一个 LR Parsing Table。但是，对于写语法分析程序来说，这个过程是必要的吗？不是的。因为我们已经构造出了 LR_DFA，所以我们可以直接利用 LR_DFA 来完成 LR Parsing Table 的所有功能。在我自己定义的 LR_DFA 中，每个状态的出边上都有符号和动作（这个地方和龙书上类似），而考虑到 Parsing Table 的 ACTION 中的 SHIFT 动作和表中 GOTO 动作的相似性（可以把对非终结符的 GOTO 看成是 SHIFT 非终结符），所以可以将这两个状态合并。那么就可以借助 LR_DFA，通过状态间的转移来实现 LR Parsing Table 的功能了。

10. Feelings and comments

这次实验，老师提供了三种方法可供选择，我和实验一一样，也是没有选择难度较为简单的 **Hard Coding** 方法（仅仅只针对某个特定的情况计算自动机、分析表，并根据分析表来编程）；也没有选择容易编程实现的 **LL(1) Parser**（对于直接语法分析树的构建，我在实验一中也成功完成了）；而是选择了**难度最高**的方法，也就是**编写代码，使程序自动完成由文法构建 LR_DFA 再构建 LR_Parser 这个过程**。当然，在完成后，我还给自己加了个额外的任务：**解决文法的二义性问题（比如经典的悬空 else 问题）**。因为平时积累了很多写代码的经验，所以这次实验虽然对我来说是充满挑战的，但是整体我也只花了 3-4 天的时间（大致 30 个小时左右）完成。可以说，在整个完成实验的过程中，我的代码能力和对 **LR(1) Parser** 的理解提高迅速。从刚开始的对 **FIRST** 函数的实现，到计算闭包，再到完成语法分析程序，一步一步，我慢慢夯实了基础，更加深了对编译原理的兴趣。完成了所有代码的编写后，我感觉非常欣喜，感觉自己对编译原理课程的理解加深了很多很多，完成后，虽然是深夜，但我禁不住跳了起来，我为自己的成果感到骄傲。我觉得，学习就是这样一个过程，一步一步往上攀爬，走了很久以后，回头再看，会惊讶于自己已经走到了那么高的地方了，但这一共也只是很小的一段旅程，我还需继续向上，永不放弃。我觉得这次的实验鼓舞了我进一步钻研计算机科学领域的信心，而我也相信，我有能力、也有勇气在这条路上走得更远。