

天津大学

《编译原理与技术》结课报告

大作业实验报告



学	院	<u>智能与计算学部</u>
专	业	<u>软件工程</u>
年	级	<u>2022 级</u>
小	组	<u>小组 5</u>
学	号	<u>30222444(55,31,26,34,19)</u>

2024 年 11 月 23 日

目 录

第一章	文法的扩展	1
1.1	struct 新增文法	1
1.2	union 新增文法	2
1.3	switch 新增文法	2
1.4	扩展后文法	3
第二章	词法分析器	7
2.1	代码设计	7
2.2	词法分析流程图	8
2.3	代码功能实现	8
2.4	源程序编译步骤	14
第三章	语法分析器	15
3.1	语法分析器的设计	15
3.2	语法分析器的代码实现	15
3.3	语法输出格式说明	21
3.4	生成的分析表	21
3.5	源程序编译步骤	21
第四章	测试过程遇到的问题	22
4.1	集中测试流程	22
4.2	词法分析测试中的问题	22
4.3	词法分析问题解决方案	22
4.4	语法分析测试中的问题:	22
4.5	语法分析问题解决方案:	23

第五章	成员分工介绍	24
第六章	补充说明	25

第一章 文法的扩展

对于原本给定的 C-文法, 我们在此基础上添加实现 struct, union, switch 的文法.

1.1 struct 新增文法



```
1  compUnit -> structDef compUnit // struct类型定义, 与函数定义、变量定义同级
2  structDef -> struct structType { structBlockElem } structVar ; // struct定义块入口
3  structType -> Ident // struct类型
4  structBlockElem -> decl structBlockElem // struct成员变量
5  structBlockElem -> funcDef structBlockElem // struct成员函数
6  structBlockElem -> $ // struct块内元素结束
7  structVar -> Ident argStructVar // struct类型变量, 紧跟struct类型定义后声明
8  structVar -> $ // 无跟随变量定义
9  argStructVar -> , Ident argStructVar // struct类型变量多重声明, 类似a, b
10 argStructVar -> $ // struct类型变量跟随多重声明结束
11 decl -> structDecl // struct类型变量声明入口
12 structDecl -> struct structType Ident argStruct ; // struct类型变量声明
13 argStruct -> , Ident argStruct // 扩展声明, 类似struct ST a, b;
14 argStruct -> $ // 结束变量扩展
15 funcFParam -> struct structType Ident // 函数声明中struct类型变量参数
```

图 1-1 struct 新增文法

- 校验记录: 测试文档中给定样例时报错, 发现无法识别函数定义中 struct 类型参数。
- 添加新文法: `funcFParam -> struct structType Ident`
- 实现对函数参数的 struct 类型识别。

1.2 union 新增文法

```
union新增文法
Plain Text Yuque Light
1 compUnit -> unionDef compUnit // union类型定义, 与函数定义、变量定义同级
2 unionDef -> union unionType { unionBlockElem } unionVar ; // union定义块入口
3 unionType -> Ident // union类型
4 unionBlockElem -> decl unionBlockElem // union成员变量
5 unionBlockElem -> $ // union块内元素结束
6 unionVar -> Ident argUnionVar // union类型变量, 紧跟union类型定义后声明
7 unionVar -> $ // 无跟随变量定义
8 argUnionVar -> , Ident argUnionVar // union类型变量多重声明, 类似a, b
9 argUnionVar -> $ // struct类型变量跟随多重声明结束
10 decl -> unionDecl // union类型变量声明入口
11 unionDecl -> union unionType Ident argUnion ; // union类型变量声明
12 argUnion -> , Ident argUnion // 扩展声明, 类似union UN a, b;
13 argUnion -> $ // 结束变量扩展
14 funcFParam -> union unionType Ident // 函数声明中union类型变量参数
```

图 1-2 union 新增文法

- 校验记录：类似 struct 文法，测试时发现函数定义时，不识别 union 类型参数。
- 添加新文法：funcFParam -> union unionType Ident
- 实现对函数参数的 union 类型识别。

1.3 switch 新增文法

```
switch新增文法
Plain Text Yuque Light
1 blockItem -> switchBlock blockItem // switch入口, 与函数内变量等同级
2 switchBlock -> switch ( unaryExp ) { caseBlock argCaseBlock defaultBlock } // switch块入口
3 caseBlock -> case constExp : { switchBlockElem } // case块入口
4 argCaseBlock -> caseBlock argCaseBlock // 多case块扩展
5 argCaseBlock -> $ // case块扩展结束
6 defaultBlock -> default : { switchBlockElem } // default块入口
7 defaultBlock -> $ // switch块不含default
8 switchBlockElem -> stmt switchBlockElem // switch块内语句
9 switchBlockElem -> $ // switch块内结束
10 stmt -> break // 添加break关键字
```

图 1-3 switch 新增文法

- 校验记录：测试给定样例时发现不识别 switch 块，检查后发现 switch 块入口应和 decl 与 stmt 同级，由 blockItem 推出，位于函数内部。

- 将源错误产生式: `compUnit -> switchBlock compUnit` 改为: `blockItem -> switchBlock blockItem`
- 实现正确识别。

1.4 扩展后文法

在添加了上述三个模块的文法之后, 我们得到了包含 `struct, union, switch` 语句块的 c-文法, 并且在新增的文法行尾, 添加了该文法的注释以便于理解:

1. `program -> compUnit`
2. `compUnit -> decl compUnit`
3. `compUnit -> funcDef compUnit`
4. `compUnit -> $`
5. `decl -> constDecl`
6. `decl -> varDecl`
7. `constDecl -> const bType constDef argConst ;`
8. `argConst -> , constDef argConst`
9. `argConst -> $`
10. `constDef -> Ident = constInitVal`
11. `constInitVal -> constExp`
12. `varDecl -> bType varDef argVarDecl ;`
13. `argVarDecl -> , varDef argVarDecl`
14. `argVarDecl -> $`
15. `varDef -> Ident argVarDef`
16. `argVarDef -> = initVal`
17. `argVarDef -> $`
18. `initVal -> exp`
19. `bType -> int`
20. `bType -> float`
21. `bType -> char`
22. `funcDef -> funcType Ident (funcFParams) block`
23. `funcType -> void`
24. `funcFParams -> funcFParam argFunctionF`
25. `funcFParams -> $`
26. `argFunctionF -> , funcFParam argFunctionF`
27. `argFunctionF -> $`
28. `funcFParam -> bType Ident`

29. block -> blockItem
30. blockItem -> decl blockItem
31. blockItem -> stmt blockItem
32. blockItem -> \$
33. stmt -> exp ;
34. stmt -> ;
35. stmt -> block
36. stmt -> return argExp ;
37. argExp -> \$
38. argExp -> exp
39. exp -> assignExp
40. lVal -> Ident
41. primaryExp -> (exp)
42. primaryExp -> number
43. number -> INT
44. unaryOp -> +
45. unaryOp -> -
46. unaryOp -> !
47. unaryExp -> unaryOp unaryExp
48. unaryExp -> Ident callFunc
49. callFunc -> (funcRParams)
50. callFunc -> \$
51. unaryExp -> primaryExp
52. funcRParams -> funcRParam argFunctionR
53. funcRParams -> \$
54. argFunctionR -> , funcRParam argFunctionR
55. argFunctionR -> \$
56. funcRParam -> exp
57. mulExp -> unaryExp mulExpAtom
58. mulExpAtom -> * unaryExp mulExpAtom
59. mulExpAtom -> / unaryExp mulExpAtom
60. mulExpAtom ->
61. mulExpAtom -> \$
62. addExp -> mulExp addExpAtom
63. addExpAtom -> + mulExp addExpAtom
64. addExpAtom -> - mulExp addExpAtom

65. addExpAtom -> \$
66. relExp -> addExp relExpAtom
67. relExpAtom -> < addExp relExpAtom
68. relExpAtom -> > addExp relExpAtom
69. relExpAtom -> <= addExp relExpAtom
70. relExpAtom -> >= addExp relExpAtom
71. relExpAtom -> \$
72. eqExp -> relExp eqExpAtom
73. eqExpAtom -> == relExp eqExpAtom
74. eqExpAtom -> != relExp eqExpAtom
75. eqExpAtom -> \$
76. assignExp -> eqExp assignExpAtom
77. assignExpAtom -> = eqExp assignExpAtom
78. assignExpAtom -> \$
79. constExp -> assignExp
80. compUnit -> structDef compUnit // struct 类型定义，与函数定义、变量定义同级
81. structDef -> struct structType structBlockElem structVar ; // struct 定义块入口
82. structType -> Ident // struct 类型
83. structBlockElem -> decl structBlockElem // struct 成员变量
84. structBlockElem -> funcDef structBlockElem // struct 成员函数
85. structBlockElem -> \$ // struct 块内元素结束
86. structVar -> Ident argStructVar // struct 类型变量，紧跟 struct 类型定义后声明
87. structVar -> \$ // 无跟随变量定义
88. argStructVar -> , Ident argStructVar // struct 类型变量多重声明，类似 a, b
89. argStructVar -> \$ // struct 类型变量跟随多重声明结束
90. decl -> structDecl // struct 类型变量声明入口
91. structDecl -> struct structType Ident argStruct ; // struct 类型变量声明
92. argStruct -> , Ident argStruct // 扩展声明，类似 struct ST a, b;
93. argStruct -> \$ // 结束变量扩展
94. funcFParam -> struct structType Ident // 函数声明中 struct 类型变量参数
95. compUnit -> unionDef compUnit // union 类型定义，与函数定义、变量

定义同级

96. unionDef -> union unionType unionBlockElem unionVar ; // union 定义块入口
97. unionType -> Ident // union 类型
98. unionBlockElem -> decl unionBlockElem // union 成员变量
99. unionBlockElem -> \$ // union 块内元素结束
100. unionVar -> Ident argUnionVar // union 类型变量, 紧跟 union 类型定义后声明
101. unionVar -> \$ // 无跟随变量定义
102. argUnionVar -> , Ident argUnionVar // union 类型变量多重声明, 类似 a, b
103. argUnionVar -> \$ // struct 类型变量跟随多重声明结束
104. decl -> unionDecl // union 类型变量声明入口
105. unionDecl -> union unionType Ident argUnion ; // union 类型变量声明
106. argUnion -> , Ident argUnion // 扩展声明, 类似 union UN a, b;
107. argUnion -> \$ // 结束变量扩展
108. funcFParam -> union unionType Ident // 函数声明中 union 类型变量参数
109. blockItem -> switchBlock blockItem // switch 入口, 与函数内变量等同级
110. switchBlock -> switch (unaryExp) caseBlock argCaseBlock defaultBlock // switch 块入口
111. caseBlock -> case constExp : switchBlockElem // case 块入口
112. argCaseBlock -> caseBlock argCaseBlock // 多 case 块扩展
113. argCaseBlock -> \$ // case 块扩展结束
114. defaultBlock -> default : switchBlockElem // default 块入口
115. defaultBlock -> \$ // switch 块不含 default
116. switchBlockElem -> stmt switchBlockElem // switch 块内语句
117. switchBlockElem -> \$ // switch 块内结束
118. stmt -> break // 添加 break 关键字

第二章 词法分析器

2.1 代码设计

2.1.1 设计思路

我们需要对待编译的代码进行分析，需要识别代码中的字符组合并将他们划分到不同的类型范畴之中，这里对于字符的识别我们首先将字符分为多个类别，可以通过将字符逐个读入内存并根据读入字符的类别来给当前已经读取的字符串赋予一个状态，当读取的某一个字符后字符串的状态转移到我们所期望的目标状态后即可将此时的字符串判定为一个 token，后续的工作就是记录这个 token 以方便输出结果，然后继续读取待编译的代码中后续的字符来识别出其他的 tokens。

2.1.2 功能概要

整体的代码功能区有两大部分，第一部分负责对一个 NFA 状态转换图进行处理，我们将这个图进行拆分并构建为一个由多个三元项组成的列表，然后对其进行确定化以及最小化处理，并得到最后的最小化三元组列表以方便在进行字符识别的过程中使用。第二部分负责对待编译代码的字符的读取和识别并根据第一步生成的最小化后的状态转换表进行字符的归类处理，当识别到一个 token 后将其记录并输出，后续重复进行字符识别并归类为 token 的操作直到整个代码文件被识别完毕。

2.2 词法分析流程图

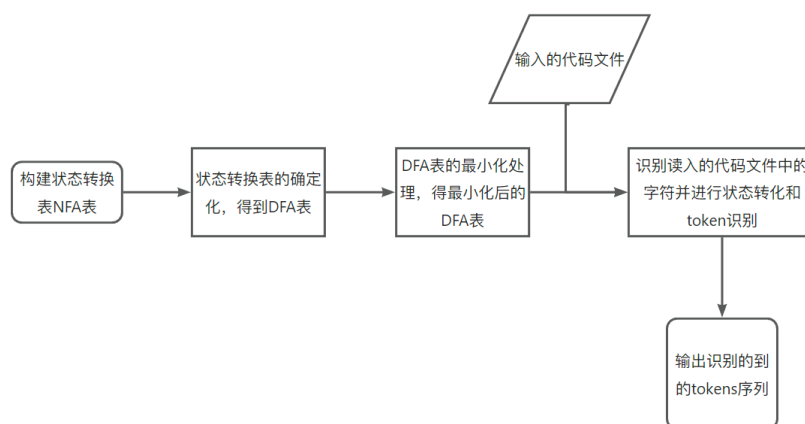


图 2-1 词法分析流程图

2.3 代码功能实现

1. 状态转换表的构建

- 根据状态转换图我可以将状态转换图进行拆分并表示为如下格式的状态转换表：

transform_table: item, item, ……, item

- 其中 item 的格式如下：

item: (a, b, c)

- a: 当前输入的字符类型
- b: 当前状态
- c: 当前状态添加上当前输入字符后的状态

- 同时这里对读取待编译的代码时输入的字符进行了分类，设置了符号表 symbols_table 并对不同的字符以及状态进行了编码，编码表为 lex_state_labels。

注：具体代码见 data_deal.py

下图展示了由转换表生成的 NFA 图，用于描述一系列状态之间基于输入符号的变化。在这张图中，每个圆形节点代表一个状态，而连接这些节点的带箭头的线段表示状态间的迁移路径。每条边上的小标签显示了导致状

态变迁的输入字符或条件。

- 在初始状态”none”下，当接收到字符’n’, ’l’, ’-’, ’s’, ’o’, ’=’, ’<’, ’>’, ’!’, ’&’, ’|’, 或者’ ’时，分别进入相应的子状态。
- 这些子状态进一步细化了输入序列的解析，并最终导向诸如”INT”, ”FLOAT”, ”I&K”, ”SE”, ”OP” 和”CHAR” 等终结状态。

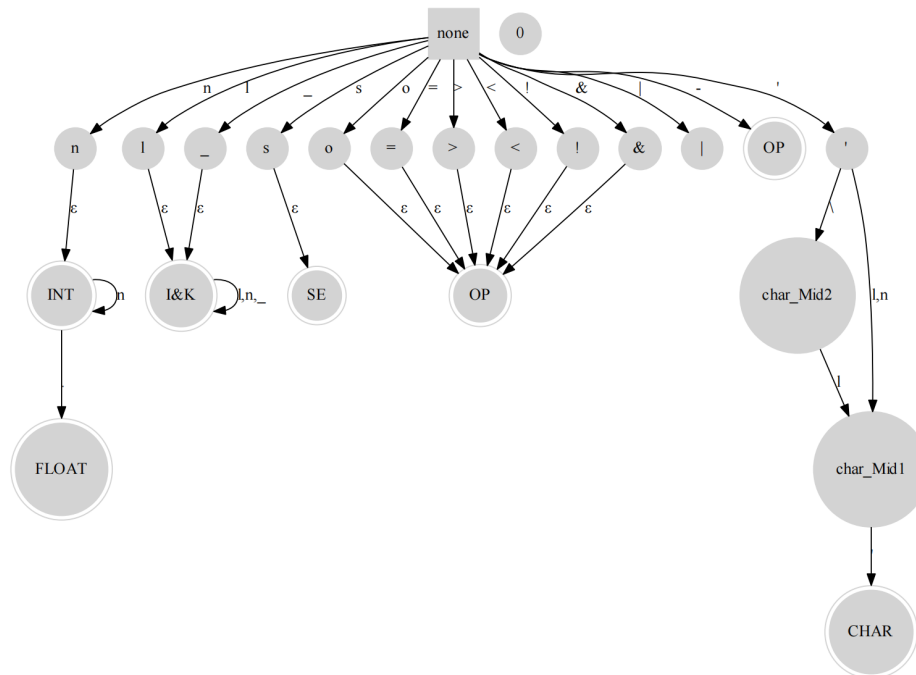


图 2-2 nfa_diagram

2. NFA 的确定化

通过将 NFA 确定化为 DFA，可以显著提高词法分析器的效率、简化实现、确保正确性和存储效率。同时 DFA 的确定性特性使得词法分析器在处理输入时的行为更加可预测和一致，避免了 NFA 中可能出现的歧义问题。因此，我们需要先对 NFA 进行确定化操作。

(a) 初始化 DFA

- 首先创建一个新的 DFA 对象，用于储存 DFA 的所有信息，包括状态、输入符号、初始状态、最终状态、状态标签和状态转换表。初始化一个空队列 `processing_set` 用于存储待处理的状态集合，初始化一个空字典 `state_to_id` 用于储存状态集合到 ID 的映射，初始化一个整数 `next_id` 用于生成新

的状态 ID。

- 设置 DFA 的输入符号为 NFA 的输入符号集，设置 DFA 的初始状态为 1, 初始状态集合 1。

(b) 主循环

while 循环：处理 processing_set 中的所有状态集合。

- current_states: 从 processing_set 中取出一个状态集合。
- current_id: 获取当前状态集合对应的 DFA 状态 ID。
- for 循环：遍历所有输入符号 ch。
- next_states: 计算当前状态集合在输入符号 ch 下的下一状态集合。
- if next_states 为空，跳过当前迭代。
- if next_states 未被处理过：next_id: 生成新的状态 ID。将 next_states 映射到新的状态 ID。

将新的状态 ID 添加到 DFA_states 中。

将 next_states 添加到 processing_set 中。

检查 next_states 是否包含 NFA 的最终状态，如果是，则将新的状态 ID 标记为 DFA 的最终状态，并设置相应的状态标签。

- 更新 DFA.trans_map，记录从当前状态到下一状态的转换。

(c) 最后返回构建好的 DFA 对象

注: 详细见代码 dfa.py

下图展示了一个 NFA 经过确定化所形成的 DFA 的状态转换图，消除了 NFA 中固有的不确定性，使其成为一种更易于理解和实施的形式。在这个图中，每个圆圈代表一个状态，而箭头则表示从一个状态到另一个状态的转移路径。具体来说：

- 圆圈内的数字或符号标识了不同的状态。
- 箭头上方的的小标签指示了触发状态转移的输入字符或条件。
- 例如，状态”1” 可以通过输入字符’s’ 转移到状态’SSE’，也可以通过输入字符’&’ 转移到状态’INT’ 等等。

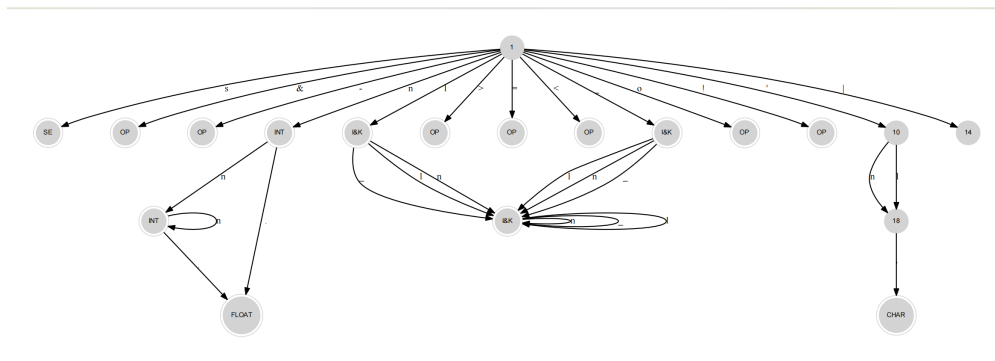


图 2-3 dfa_diagram

3. DFA 的最小化

最小化 DFA 可以去除冗余状态，使自动机更加简洁和高效。最小化后的 DFA 是唯一的，这意味着对于同一个正则表达式，无论从哪个 NFA 确定化而来，最终的最小化 DFA 都是相同的。DFA 最小化不仅提高了词法分析器的效率，还简化了实现，确保了正确性和唯一性。

(a) 初始化最小化 DFA:

创建一个新的 DFA 对象 `min_DFA`, 用于储存最小化后的 DFA, 初始化一个空字典 `state_mapping`, 用于储存状态映射。初始化两个空集合 `non_final_states` 和 `final_states_by_label`, 分别用于储存非最终状态和最终状态。

(b) 区分非最终状态和最终状态:

- for state in DFA.states; //遍历 DFA 的所有状态
- if 状态是最终状态, 将其添加到 `final_states_by_label` 集合中。
- else 将其添加到 `non_final_states` 集合中。

(c) 分配新的状态 ID:

- 初始化一个新的状态 ID 计数器 `new_state_id`。
- 遍历所有非最终状态, 为其分配新的状态 ID, 并将映射关系存储在 `state_mapping` 中。
- 遍历所有最终状态, 为其分配新的状态 ID, 并将映射关系存储在 `state_mapping` 中。

(d) 构建最小化 DFA 的状态, 初始状态和最终状态:

- 遍历 `state_mapping` 中的所有状态映射, 将新的状态 ID 添加到 `min_DFA_states` 中。

- 如果原始状态是初始状态，将新的状态 ID 添加到 `min_DFA.start` 中。
- 如果原始状态是最终状态，将新的状态 ID 添加到 `min_DFA.final` 中，并设置相应的状态标签。

(e) 构建最小化 DFA 的状态转换表：

- 遍历 DFA 的状态转换表 `trans_map`。
- 获取当前状态转换表条目的起始状态和目标状态在最小化 DFA 中的新状态 ID。
- 将新的状态转换表条目添加到 `min_DFA.trans_map` 中。

(f) 返回最小化后的 DFA

注：详细代码见 `minimize_dfa.py`

图中展示了一个 DFA 经过最小化算法后生成的新的 DFA，通过合并功能相同或相似的状态，不仅可以降低存储开销，还能加快运行速度。它具有多个状态和转换规则，每个灰色圆圈代表一个状态，而带有箭头的线条表示从一个状态到另一个状态的过渡。这些过渡由特定的输入字符触发，如图中标注的字母和符号所示。

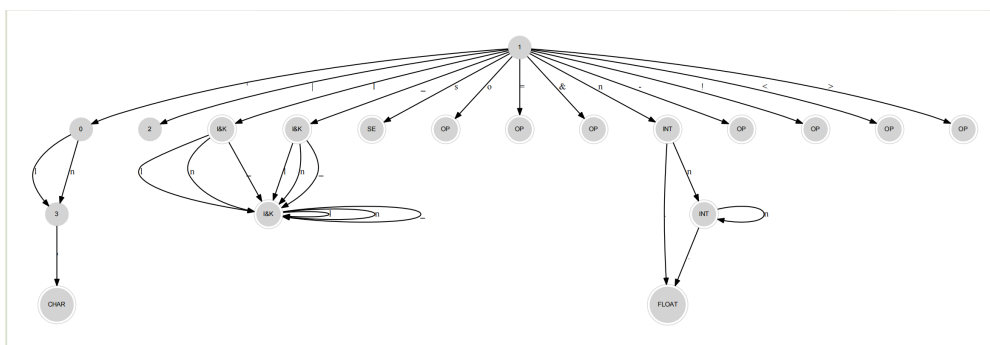


图 2-4 minimized_dfa_diagram

4. lexer 分析

我们在这一步对待编译的代码进行按行读入，对于每一行的字符进行 `tokens` 的识别和输出，伪代码如下：

```
for line in code do //code为输入的待编译的代码
    for each_char in line do //遍历一行中的每一个字符
        if each_char != ' ': //跳过空格
```

```

    获取当前字符的类型 char_type
    matched = False // 标记当前是否别到一个可以进行状态转移的状态
    for each_item in transform_map: // 遍历状态转换表
        if 表中找到找到一个输入类型和当前状态匹配的转换关系:
            进行状态转移并记录当前读入的字符
            matched = True // 标记识别到可转移的状态
            break
    if matched do
        if 当前状态是终态 do
            next_state = none
            next_char
            for each_item in transform_map do
                if 在转换表中找到匹配的 do
                    next_state = each_item.next_state // 进行状态转移
            if next_state == none 或者 next_state 是终态:
                记录并输出当前识别到的 token
                重置记录读入字符的变量
                重置当前状态

```

详细代码见 lexer_utils.py

5. tokens 生成

(a) tokens 输出格式

输出格式上, 我们严格按照大作业中规定的格式, 我们对 tokens 的输出格式进行了规范处理。

Token 输出格式:

[待测代码中的单词符号] [TAB] <[单词符号种别],[单词符号内容]>

- 其中, 单词符号种别为 KW (关键字)、OP (运算符)、SE (界符)、IDN (标识符) INT (整形数)
- 单词符号内容 KW、OP、SE 为其编号
- 其余类型为其值

假设我们需要分析的代码是:

```

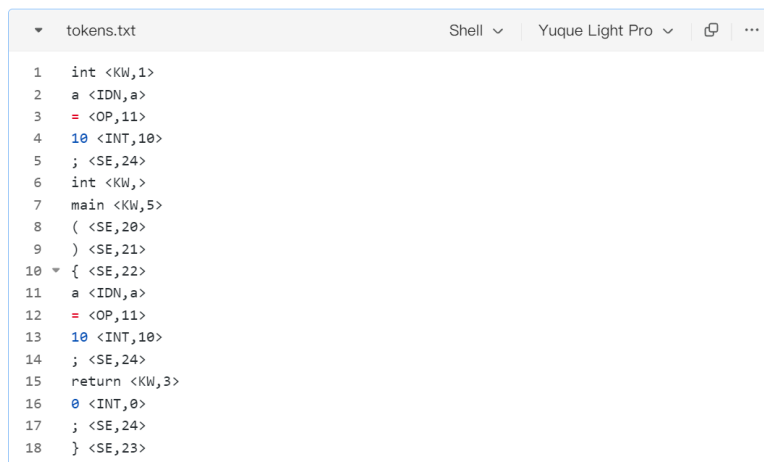
1  int a = 10;
2  int main() {
3      a=10;
4      return 0;

```



```
5  }
```

词法分析器生成的 tokens 序列如下:



```
tokens.txt
1  int <KW,1>
2  a <IDN,a>
3  = <OP,11>
4  10 <INT,10>
5  ; <SE,24>
6  int <KW,>
7  main <KW,5>
8  ( <SE,20>
9  ) <SE,21>
10 { <SE,22>
11 a <IDN,a>
12 = <OP,11>
13 10 <INT,10>
14 ; <SE,24>
15 return <KW,3>
16 0 <INT,0>
17 ; <SE,24>
18 } <SE,23>
```

图 2-5 tokens 序列

生成的词法分析器的结果保存在 output.lex_output 中

(b) 代码实现

对于 tokens 的生成, 利用在 lexer 分析的过程中所识别到的字符串, 在字符映射表 symbols 和符号表 symbols_table 中进行查询, 以获取识别到的字符串所对应的编号和符号类型, 然后使用字符串拼接的方法生成语法分析器所需要格式的 tokens 序列, 具体代码实现见 helper_func.py 的 get_tokens 函数。

2.4 源程序编译步骤

本语法分析器使用 python 进行编程, 在 python 环境下, 可以直接在 src 目录下执行 python Lexer/lexer.py 文件即可执行脚本

第三章 语法分析器

3.1 语法分析器的设计

如图, 根据 LL(1) 分析法的原则, 我们的文法分析器将读入的文法进行预处理, 考虑到 LL(1) 分析的条件: 无左递归, 无二义性, 有限的向前看。

于是语法分析器的实现应该按照如下步骤:

1. 消除文法中可能存在的左递归,
2. 利用消除左递归之后的文法构建 FIRST 与 FOLLOW 集合,
3. 得到 FIRST 与 FOLLOW 集合之后即可根据 LL(1) 分析表的构造算法构造分析表,
4. 最后结合词法分析器得到的 tokens 进行移进规约分析

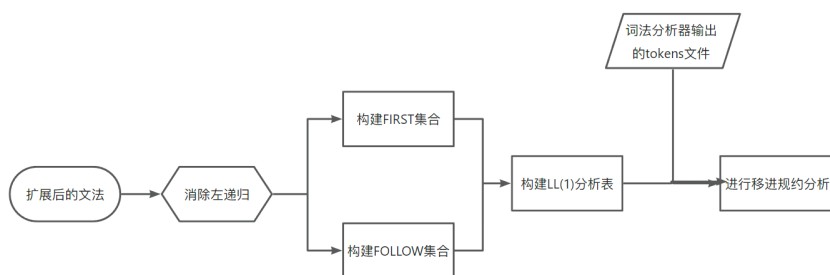


图 3-1 语法分析器的设计

3.2 语法分析器的代码实现

3.2.1 文法储存单元

对于文法的存储, 我们设计了 Grammar 用来存储扩展之后的语法规则

- Grammar 类将语法规则组织成一个字典结构, 左侧非终结符作为键, 右侧产生式列表作为值。这种结构便于快速查找和管理语法规则。
- 一些与文法密切相关的操作封装在文法类中, 便于对于文法信息的读取, 使得项目进行过程中, 小组成员不需要理解内部具体的实现细节, 有利于合作开发。
- 文法类中重写了 `__str__` 方法, 便于后续 debug 过程中便捷查看以及验证文法的正确性, 在后续的大作业实践中体现了这一点利好。

3.2.2 token 存储单元

- 在语法的移进规约分析过程中, 需要结合词法分析器输出的 tokens 文件, 对于每一条 token 信息, 我们采用 LexerToken 进行存储 (为了和 python 自带的关键字 Token 区分)。
- LexerToken 中成员变量 type,value,id 存储了 token 的必要信息。
- 封装管理单条目 token 便于后续的开发。

3.2.3 消除左递归

在项目中实现 LeftRecursionEliminator 类。接受 Grammar 类的对象作为输入, 并 z 最终得到一个消除了左递归的新的 Grammar 对象。类中包含一个主要的 eliminate__left__recursion 方法和多个辅助方法。

- eliminate__left__recursion 方法: 这个方法将所有非终结符有序排列, 并遍历所有非终结符, 对于每一非终结符, 如果一个非终结符 P_i 的产生式以另一个非终结符 P_j 开始, 则会通过 __expand_grammar 方法扩展文法, 在 P_i 所有文法扩展完成后, 使用 __eliminate_direct_left_recursion_for_one_rule 方法消除所有关于 P_i 的直接左递归。并在所有非终结符遍历完成后, 使用 __remove_useless_productions 方法删除无用的产生式。
- __expand_grammar 方法: 这个方法用于扩展文法规则, 以将间接左递归转换为直接左递归。它将一个非终结符 P_j 的所有产生式扩展到另一个非终结符 P_i 的所有以 P_j 开头的产生式中, 将所有可能的间接左递归转换为直接左递归, 便于后续处理。
- __eliminate_direct_left_recursion_for_one_rule 方法: 这个方法用于消除直接左递归。将左递归的产生式和非左递归的产生式分开处理, 并引入新的非终结符来消除左递归。例如: 对于直接左递归的产生式, $P \rightarrow P \mid$ 转换为 $P \rightarrow P'$ 和 $P' \rightarrow P'$ |
- __remove_useless_productions 方法: 这个方法用于删除那些不可达的非终结符, 即那些不能从起始符号推导出的非终结符, 从起始符号遍历所有非终结符, 修改其可达性, 直到所有非终结符的可达性确定, 消除那些不可达的产生式。

3.2.4 计算 FIRST 集合和 FOLLOW 集合

在语法分析的任务中，我们为了构造分析表，需要先计算上下文无关文法的 FIRST 集合和 FOLLOW 集合。

- 构造时需要传入的参数：
 - grammar: Grammar 类的实例，包含文法的产生式规则。
 - space_symbol: 空串符号，通常用 '\$' 表示。
 - start_symbol: 文法的开始符号。
 - end_symbol: 结束符号，通常用 '#' 表示，用于表示输入结束。
- 类中的核心方法：
 1. compute_first(self, grammar, space_symbol): 计算非终结符的 FIRST 集合。

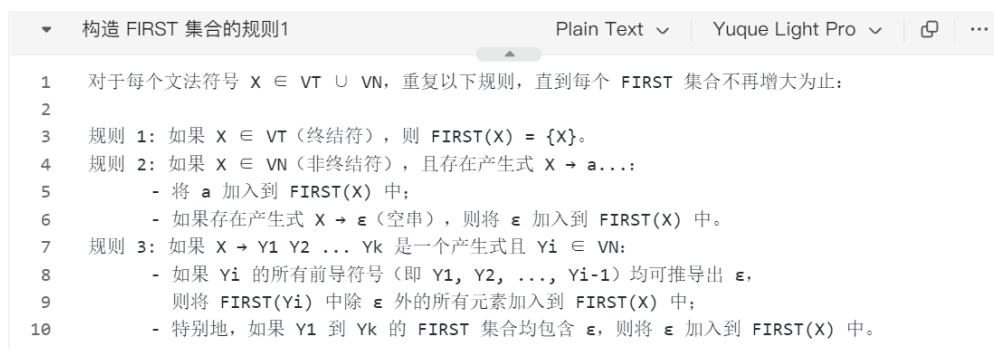


图 3-2 构造 first 集合的规则 1

实现思路:

- (a) 对于文法的非终结符，初始化所有非终结符的 FIRST 为一个空 Set
- (b) 依次遍历文法中的非终结符和产生式，取出其中的产生式
- (c) 如果产生式第一个符号是终结符，则将该终结符直接加入当前非终结符的 FIRST 集合中
- (d) 如果产生式第一个符号是非终结符，并且该非终结符的 FIRST 集合中不包含空符号，那么直接将该非终结符的 FIRST 集合除去空符号加入当前非终结符的 FIRST 集合中
- (e) 如果产生式第一个符号是非终结符，并且该非终结符的 FIRST 集合中包含空符号，那么将该非终结符的 FIRST 集合除去空符号加入当前非终结符的 FIRST 集合中，并且继续向后遍历产生式的符号，直到产生式的最后一个符号，如

果此时当前非终结符的 FIRST 集合没有空符号，将空符号加入当前非终结符的 FIRST 集合

(f) 如果产生式为 ϵ ，往当前非终结符的 FIRST 集合加入空符号

2. `compute_production_first(self, grammar, space_symbol)`: 计算产生式的 FIRST 集合。

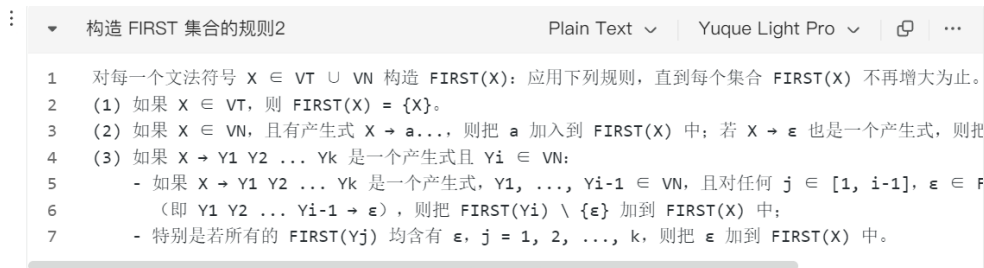


图 3-3 构造 first 集合的规则 2

实现思路:

- (a) 遍历所有产生式，初始化所有产生式的 FIRST 集合为空
- (b) 如果产生式为 ϵ ，当前产生式的 FIRST 集合添加空符号
- (c) 遍历产生式中的符号，如果第一个符号为终结符，当前产生式的 FIRST 集合添加该终结符
- (d) 如果第一个符号为非终结符，将其 FIRST 集合加入当前产生式的 FIRST 集合。如果该非终结符的 FIRST 集合中有空串，往后遍历下一个符号

3. `compute_follow(self, grammar, first_sets, space_symbol, start_symbol, end_symbol)`: 计算 FOLLOW 集合。



图 3-4 构造 follow 集合的算法

- (a) 初始化所有非终结符的 FIRST 为空 Set，其中开始符号的 FIRST 集合包含 end_symbol
- (b) 依次遍历所有非终结符和对应产生式，取出其中的产生式，遍历产生式的每个符号
- (c) 如果当前符号是非终结符，检查后续符号
- (d) 如果后续符号是终结符，则将其直接加入当前符号的 FOLLOW 集，并终止检查
- (e) 如果后续符号是非终结符，则将其 FIRST 集合（去掉空字符 ϵ ）加入当前符号的 FOLLOW 集
- (f) 如果后续符号的 FIRST 集包含空字符 ϵ ，则继续检查下一个符号
- (g) 如果当前符号是产生式的最后一个符号，或者后续所有符号的 FIRST 集都包含空字符，则将左部非终结符的 FOLLOW 集加入当前符号的 FOLLOW 集
- (h) 如果新的 FOLLOW 集有元素加入，则标记 `changed = True`，继续下一轮迭代

3.2.5 分析表的实现

项目中，为了实现语法分析，借助 LL(1) 分析法的分析过程，我们设计了 ParsingTable 来构建并存储分析表，

构造时要求传入参数：

- first_set: 字典，保存 FIRST 集合
- follw_set: 字典，保存 FOLLOW 集合
- grammar: 文法对象

根据 LL(1) 算法^[1], ParsingTable 提供了 `get_parsing_table()` 方法构建分析表

分析表构造算法		Plain Text	Yuque Light	🔗	...
1	(1) 对文法G的每个产生式 $A \rightarrow \alpha$ ，执行第(2)和(3)步；				
2	(2) 对每个终结符 $a \in \text{FIRST}(\alpha)$ ，把 $A \rightarrow \alpha$ 加入 $M[A, a]$ 中；				
3	(3) 若 $\epsilon \in \text{FIRST}(\alpha)$ ，则对任何 $b \in \text{FOLLOW}(A)$ ，把 $A \rightarrow \epsilon$ 加入 $M[A, b]$ 中；				
4	(4) 把所有无定义的 $M[A, a]$ 标上“出错标志”。				

图 3-5 分析表构造算法

伪代码表示如下:

```

1  for A ->    in grammars do
2      for a in FIRST( ) do
3          M[A,a]  = A -> alpha
4      if $ in FIRST( ) do
5          for b in FOLLOW(A)
6              M[A,b] = A -> $

```

- 通过 `get_production_from_table(L,r)` 方法, 即可获得非终结符 L, 与终结符 r 索引在分析表中对应的产生式或者是错误标记的内容
- 通过 `get_parsing_table()` 方法可以直接获取分析表
- 为了便于调试以及查看分析表内容, `ParsingTable` 提供了 `print_parsing_table()` 方法来快速打印分析表内容

3.2.6 规约序列的生成

语法分析的过程中, 我们可以根据分析表生成对应的规约序列。

构造时需要传入的参数:

- `parsing_table`: 预测分析表 (非终结符, 终结符) 对应的产生式
- `non_terminals`: 文法的非终结符集合
- `terminals`: 文法的终结符集合

实现思路:

1. 初始化输入 `token` 列表, 结尾添加结束符 `#`
2. 初始化栈, 栈顶元素为 `#`, 栈底为开始符号
3. 初始化规约序列号 `step_number`
4. 循环依次取出栈顶元素 `top` 和当前输入符号 `current_input`
5. 如果栈顶为非终结符, 根据 `(top, current_input)` 从解析表中查找产生式, 如果未找到匹配的产生式, 抛出语法错误。如果找到产生式, 将其右部逆序入栈 (空串 `$` 不入栈), 记录操作为 "reduction", 表示当前步骤进行了归约。
6. 如果栈顶为终结符, 如果栈顶元素 `top` 是终结符且与当前输入符号匹配, 弹出栈顶元素, 并移动输入指针 `index`。
7. 如果栈顶元素为 `#` 或 "EOF", 则说明输入匹配完成, 操作为 "accept"。否则, 操作为 "move", 表示成功匹配终结符。如果不匹配, 抛出语法错误。
8. 当栈为空时, 解析结束。返回记录的所以步骤

3.3 语法输出格式说明

我们的语法分析器严格按照实验指导书的规定进行输出, 每一条动作操作都遵照如下格式:

[序号] [TAB] [栈顶符号]#[面临输入符号] [TAB] [执行动作]

其中, 执行动作为 “reduction” (归约), “move” (LL(1) 分析的跳过), “accept” (接受) 或 “error” (错误)。

语法分析的结果以文本文件的形式保存在 `output.parser_output` 中。

3.4 生成的分析表

我们的词法分析器在每次执行是, 可以通过设置 `config.py` 中的 `VISUALIZE_PARSING_TABLE` 参数来设置是否构建分析表的可视化图表。

由于表格数据众多这里的图片仅供预览



图 3-6 生成的分析表

我们生成的分析表因为太大不便于在报告中显示, 可以点击[这里](#)直接查看我们生成的分析表内容。

分析表中, 空的部分表示” 出错标志”, 为了显示美观, 这里留空展示。

3.5 源程序编译步骤

本语法分析器使用 python 进行编程, 在 python 环境下, 可以直接在 `src` 目录下执行 `python Syntax/parser.py` 文件即可执行脚本

第四章 测试过程遇到的问题

4.1 集中测试流程

1. 使用 `syntax_util` 类的 `load_from_file` 函数读取文法；
2. 使用 `LeftRecursionEliminator` 类消除左递归，得到消除左递归后的文法；
3. 将新文法传入 `FirstAndFollow` 类，构建 FIRST 和 FOLLOW 集；
4. 使用 `ParsingTable` 类构建分析表，准备进行规约分析；
5. 使用 `syntax_util` 类的 `load_tokens` 函数读取词法分析结果；
6. 使用 `PredictiveParser` 类进行规约分析

4.2 词法分析测试中的问题

1. 在识别 `tokens` 的过程中出现了状态转移有误的问题，即在连续识别 “=” 和 `digit`(数字) 时词法分析器会将 “=” 和 `digit` 作为整体识别为 `IDN`。
2. 在编写 `char` 类型的识别的时候, 只实现了对数字和字母的识别, 后经查找资料^[2] 了解到 `char` 类型中可以包含转义字符, 于是我们添加了 `char` 类型中对转义字符的识别。

4.3 词法分析问题解决方案

1. 经过检查我们了解到了在识别到一个 “=” 即 `OP` 后直接读取下一个 `digit` (数字) 会进入一个错误的状态。于是我们增加了一个新的 `OP` 类型的状态, 即状态转换表中一共有两个 `OP` 类型的状态, 这样既解决了识别双字符 `OP` (“==”, “<=”, “>=”, “!=”) 状态转移的问题, 又解决了 “=” 和整型字符识别错误的问题。
2. 在状态列表 `lex_state_labels` 中新增一个状态 `char_Mid2` 来表示转移字符诸如换行符以及制表符中的转义符, 然后在状态转移表中添加相关转移三元组来解决这个问题。

4.4 语法分析测试中的问题:

1. 在语法分析的过程中, 开始我们并没有处理导致语法错误的符号, 而是直接抛出异常。进而导致出现语法错误的时候无法输出规约序列。
2. 发现每次运行程序结果不同, 且有同一产生式错误识别情况。
3. `struct` 测试文档中给定样例时报错, 发现无法识别函数定义中 `struct` 类型参数。
4. `union` 类似 `struct` 文法, 测试时发现函数定义时, 不识别 `union` 类型参数。
5. `switch` 测试给定样例时发现不识别 `switch` 块, 检查后发现 `switch` 块入口

应和 decl 与 stmt 同级，由 blockItem 推出，位于函数内部。

4.5 语法分析问题解决方案：

1. 在遇到语法错误抛出异常时，捕获异常，同时将当前符号的 action 设置为"error"，用来标记它导致语法错误的符号，将当前符号添加到规约序列中，最后返回规约序列。
2. 溯源发现分析表有误，将表中每一项填写时改为直接复制，覆盖源有条目。
3. 添加新文法：funcFParam \rightarrow struct structType Ident，实现对函数参数的 struct 类型识别。
4. 添加新文法：funcFParam \rightarrow union unionType Ident，实现对函数参数的 union 类型识别。
5. 将源错误产生式：compUnit \rightarrow switchBlock compUnit，改为：blockItem \rightarrow switchBlock blockItem，实现正确识别

第五章 成员分工介绍

- **覃邱维: 项目负责人**

负责成员调度任务安排, 负责整体的项目架构设计, 实现语法分析器的分析表构建以及可视化部分, 负责语法分析器中实体类的构造以及数据的读入, 负责展示 ppt 设计与实现, 负责语法分析器的总体设计, 负责新增文法的审查与校验. 负责实验报告的组织与撰写.

- **王俊哲: 项目成员**

负责新增文法的编写与校验. 语法分析器部分代码编写, 实现语法分析器中消除左递归的功能, 以及对语法分析器部分 bug 的修复.

- **李亮克: 项目成员**

负责词法分析器大部分代码的编写, 负责部分 NFA 状态转换表的构建, NFA 的确定化以及 DFA 的最小化的实现, 同时为词法分析器编写了测试代码.

- **聂哲浩: 项目成员**

负责语法分析器部分代码的编写, 实现语法分析器中 FIRST 和 FOLLOW 集合的计算, 以及输出规约序列的功能. 同时完成对语法分析模块的代码测试.

- **张浩然: 项目成员**

负责部分词法分析器的代码编写, 部分文档撰写及排版工作.

第六章 补充说明

本次大作业采用 python 开发, 具体的架构信息请阅读我们的项目README文件

参考文献

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. 程序设计语言编译原理 (第 3 版). 机械工业出版社, 2007. ISBN: 9787111206207.
- [2] weixin_39524439. C 语言怎么用 char 输出多个字母,c 语言中 char 类型如何存放多个字符. CSDN 博客. [在线]: 访问时间 2024 年 11 月 19 日. 链接: https://blog.csdn.net/weixin_39524439/article/details/116984201.