

# Yacc实验报告

学号： 171250649 姓名： 赵文祺

本次实验使用Kotlin语言实现，并使用Gradle进行测试和管理

## Yacc实验报告

学号： 171250649      姓名： 赵文祺  
本次实验使用Kotlin语言实现，并使用Gradle进行测试和管理  
截图

- 源文件目录截图
- GCF.y资源文件截图
- 输入文件code1.txt截图
- 输入Token截图(input1.txt)
- 输出规约表达式的截图 (output1.txt)
- 1.实现LRParser需要的具体步骤
- 2.重要的转化步骤的实现
  - 2.1 从CGF.y中读取上下文无关文法
    - 2.1.1 确定终结符与非终结符
  - 2.2 获得拓展文法
  - 2.3 根据拓展文法构造整个LR项集
    - 2.3.1 first()方法 (详见Grammar.kt)
    - 2.3.2 closure()方法 (详见LRItemSet.kt)
    - 2.3.3 goto()方法 (详见LRItemSet.kt)
  - 2.4 由LR项集组成的GOTO图构建预测分析表
    - 2.4.1 建立GOTO图
    - 2.4.2 建立预测分析表
  - 2.5 根据预测分析表进行规约(详见analyzeTable.kt)
- 3 Error handling
  - 3.1 规约Token序列
- 4 感受和实验评价

## 截图

### 源文件目录截图



▼  kotlin


▼  yacc

 AnalyzeTable.kt

 GOTOGraph.kt

 Grammar


 LRItem

 LRItemSet

 LRParser

 Main.kt

 Production

 Symbol


▶  yacc.exception

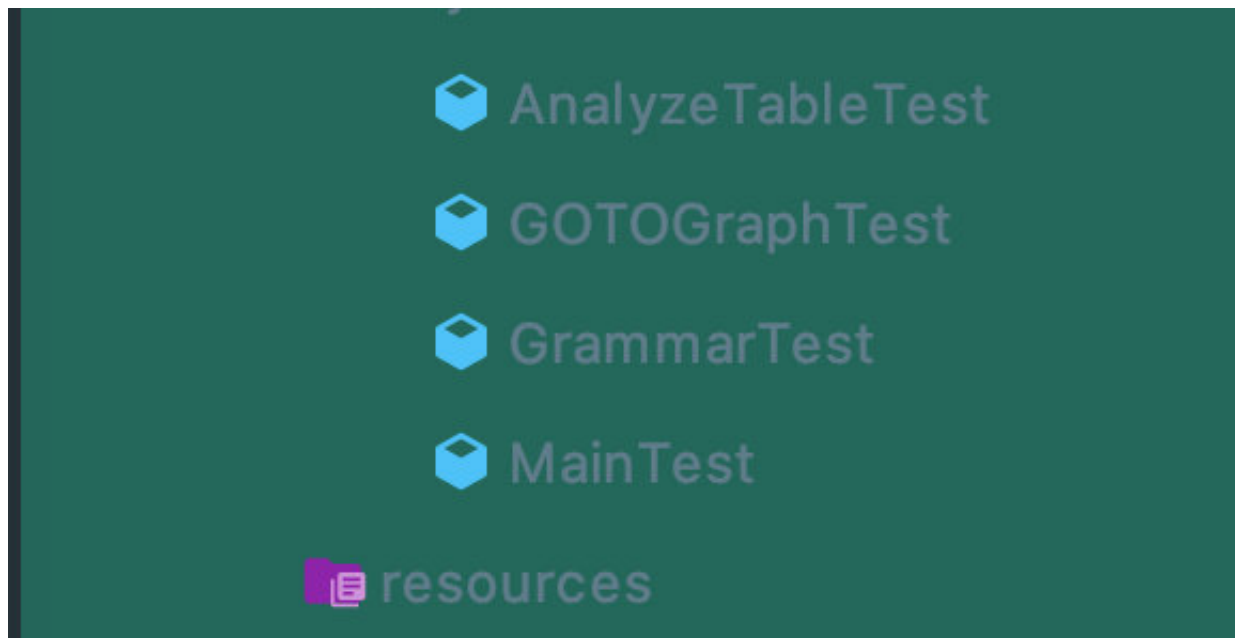
 Const.kt

 resources

▼  test

▼  kotlin

▼  yacc



## GCF.y资源文件截图

简单定义了一个支持加减乘除、括号、一元运算符、布尔运算、if/else控制语句、while控制语句的上下文无关语法，详见 GCF.y 文件

```

1  primary_expression
2      : 'id'
3      | 'number'
4      | 'literal'
5      | 'left_pare' expression 'right_pare'
6      ;
7
8  unary_expression
9      : primary_expression
10     | 'add_op' unary_expression
11     | 'unary_op' unary_expression
12     ;
13
14 multiplicative_expression
15     : unary_expression
16     | multiplicative_expression 'mul_op' unary_expression
17     ;
18
19 additive_expression
20     : multiplicative_expression
21     | additive_expression 'add_op' multiplicative_expression
22     ;
23
24 shift_expression
25     : additive_expression
26     | shift_expression 'shift_op' additive_expression
27     ;
28
29 relational_expression
30     : shift_expression
31     | relational_expression 'compare_op' shift_expression

```

## 输入文件code1.txt截图

支持简单的原始类型声明，if/else，while语句

```

double a = 1 + 2.0 + 2e-10;
char num;
num = 'a';
while (a < 100) {
    if (a != 10) {
        a += 10;
    } else {
        a -= 1;
        num = 'b';
    }
}

```

## 输入Token截图(input1.txt)

通过上一次实验得到的Lex，将输入源文件转为token的形式

```
1 Token{pattern='primitive_type', lexeme='double', id=0}
2 Token{pattern='id', lexeme='a', id=0}
3 Token{pattern='assignment', lexeme='=', id=0}
4 Token{pattern='number', lexeme='1', id=0}
5 Token{pattern='add_op', lexeme='+', id=0}
6 Token{pattern='number', lexeme='2.0', id=1}
7 Token{pattern='add_op', lexeme='+', id=1}
8 Token{pattern='number', lexeme='2e-10', id=2}
9 Token{pattern='semicolon', lexeme=';', id=0}
10 Token{pattern='primitive_type', lexeme='char', id=1}
11 Token{pattern='id', lexeme='num', id=1}
12 Token{pattern='semicolon', lexeme=';', id=1}
13 Token{pattern='id', lexeme='num', id=2}
14 Token{pattern='assignment', lexeme='=', id=1}
15 Token{pattern='literal', lexeme=' 'a', id=0}
16 Token{pattern='semicolon', lexeme=';', id=2}
17 Token{pattern='while', lexeme='while', id=0}
18 Token{pattern='left_pare', lexeme='(', id=0}
19 Token{pattern='id', lexeme='a', id=3}
20 Token{pattern='compare_op', lexeme='<', id=0}
21 Token{pattern='number', lexeme='100', id=3}
22 Token{pattern='right_pare', lexeme=')', id=0}
23 Token{pattern='left_cur_bra', lexeme='{', id=0}
24 Token{pattern='if', lexeme='if', id=0}
25 Token{pattern='left_pare', lexeme='(', id=1}
26 Token{pattern='id', lexeme='a', id=4}
27 Token{pattern='equality_op', lexeme='!=', id=0}
28 Token{pattern='number', lexeme='10', id=4}
29 Token{pattern='right_pare', lexeme=')', id=1}
30 Token{pattern='left_cur_bra', lexeme='{', id=1}
31 Token{pattern='id', lexeme='a', id=5}
```

## 输出规约表达式的截图 (output1.txt)

以空白行分割两次规约。

sentence指的是未规约之前的语句，reduce指的是此次规约所使用的产生式，result指的是最后一次产生的结果

最后将文法规约为grammar即结束

```

284 reduce: assignment_stmt -> 'id' 'assignment' bool_expression 'semicolon'
285
286 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
287 reduce: stmt -> assignment_stmt
288
289 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
290 reduce: stmts -> stmts stmt
291
292 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
293 reduce: if_else_stmt -> 'if' 'left_pare' bool_expression 'right_pare' 'left_cur_b
294
295 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
296 reduce: stmt -> if_else_stmt
297
298 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
299 reduce: stmts -> stmt
300
301 sentence: stmts 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_bra' s
302 reduce: while_stmt -> 'while' 'left_pare' bool_expression 'right_pare' 'left_cur_
303
304 sentence: stmts while_stmt $
305 reduce: stmt -> while_stmt
306
307 sentence: stmts stmt $
308 reduce: stmts -> stmts stmt
309
310 sentence: stmts $
311 reduce: grammar -> stmts
312
313 result: grammar $
314

```

## 1.实现LRParser需要的具体步骤

---

1. 从CGF.y中读取上下文无关文法
2. 获得拓展文法
3. 根据拓展文法构造整个LR项集
4. 由LR项集组成的GOTO图构建预测分析表
5. 根据预测分析表进行规约

## 2.重要的转化步骤的实现

---

### 2.1 从CGF.y中读取上下文无关文法

### 2.1.1 确定终结符与非终结符

因为采用了yacc和lex松耦合的设计，因此yacc读入流将是lex生成的Token序列，因此将所有的Token模式名都定义为终结符。在书写时，终结符需要用两个单引号包围来表示。否则表示一个非终结符。

比如 `'number'` 是一个终结符，而 `expression` 是一个非终结符。

实践中，先定义了一个密封类Symbol，以及Symbol的三个子类TerminalSymbol, NonterminalSymbol, EpsilonSymbol，这样便于使用when表达式进行Symbol的实际类型判断。

## 2.2 获得拓展文法

为了确定哪些非终结符是开始符，并添加额外的START产生式。

我才用的方法是：如果一个非终结符没有出现在除以自身为产生式左部的产生式右部中，那么这个非终结符是开始符

```
// should not exist in other production's derivation part
startProds = tempProds
    .map { it.symbol }
    .toSet()
    .filter { symbol ->
        tempProds.filter { it.symbol != symbol } // except it self
        .all { !it.derive.contains(symbol) }
    }
    .map { Production(ALL_START_SYMBOL, it.content) }
```

- 将产生式映射为左部符号的集合
- 过滤以自身为左部的产生式
- 判断是否出现在产生式的右部
- 将符合条件的符号映射到 START -> a上

这里借助kotlin的集合操作，较为简洁地实现了需求。

## 2.3 根据拓展文法构造整个LR项集

### 2.3.1 first()方法（详见Grammar.kt）

```
fun first(symbols: List<Symbol>): Set<TerminalSymbol> {
    require(symbols.isNotEmpty()) {
        "Cannot find first() for an empty list<yacc.Symbol>"
    }

    // take first symbol of list
    val firstSym = symbols.first()

    // return if first symbol is terminal symbol
```

```

        if (firstSym is TerminalSymbol) return setOf(firstSym)

        // find first(firstSym)
        val termSet = mutableSetOf<TerminalSymbol>()
        for (prod in prods.filter { it.symbol == firstSym }) {
            termSet += first(prod.derive)
        }

        // if symbol set of first symbol does not contains epsilon or this is
the last
        if (!termSet.contains(EpsilonSymbol()) || symbols.size == 1) return
termSet

        val leftTermSet = first(symbols.drop(1))

        // not all set contains epsilon, remove epsilon
        if (!leftTermSet.contains(EpsilonSymbol())) {
            termSet.remove(EpsilonSymbol())
        }
        return termSet + leftTermSet
    }
}

```

- 取出列表头部元素，判断是否是终结符，如果是则直接返回单元素的集合
- 递归调用 `first()` 方法，获得头部元素的first集合
- 如果该first集合不包含  $\epsilon$  终结符，或者这是列表中最后一个元素，直接返回first集合
- 递归调用first求除头部元素以外的first集合
- 去掉  $\epsilon$  除非所有集合都含有  $\epsilon$

### 2.3.2 closure()方法（详见LRItemSet.kt）

```

// in place function
fun closure(): LRItemSet {
    // A -> a.Bc, d
    val itemQueue: Deque<LRItem> = ArrayDeque(itemSet)
    while (itemQueue.isNotEmpty()) {
        val item = itemQueue.poll()
        val symbol = item.symbolBehindDot
        if (symbol is TerminalSymbol) continue
        // B -> y
        for (prod in grammar.prodsDeriveFrom(symbol as NonterminalSymbol))
        {
            // b in FIRST(cd)
            val symList: List<Symbol> =
                if (item.symbolAcrossDot == null)
                    listOf(item.lookAheadSymbol)
                else

```



```

        listOf(item.symbolAcrossDot!!,
item.lookAheadSymbol)
        for (termSym in grammar.first(symList)) {
            // B -> .y, b
            val newItem = LRItem(prod, termSym, 0)
            if (newItem in itemSet) continue
            addItem(newItem)
            itemQueue.add(newItem)
        }
    }
}
return this
}

```

在LRItem类中定义 `lookAheadSymbol` 来表示向前看符，定义 `symbolBehindDot` 表示点之后存在的符号，如果点在产生式的最后，则返回 `$`。

定义 `symbolAcrossDot` 表示点之后第二个符号  $A \rightarrow a.Bc$  中的  $c$ 。

在 `closure()` 方法中使用一个队列来保存待处理的产生式，如果生成的新产生式不在总的 `itemSet` 中，则将其加入。

### 2.3.3 goto()方法（详见LRItemSet.kt）

```

fun goto(sym: Symbol): LRItemSet {
    require(sym in symbolsBehindDot) {
        "$sym is not behind dot"
    }
    val itemSet = LRItemSet(grammar)
    for (item in this.itemSet) {
        if (item.symbolBehindDot == sym) {
            itemSet.addItem(LRItem(item.production, item.lookAheadSymbol,
item.dotIndex + 1))
        }
    }
    return itemSet.closure()
}

```

先用grammar初始化一个LR项集，将本LR项中能够通过sym得到的新LRItem放入新的LR项集中

## 2.4 由LR项集组成的GOTO图构建预测分析表

### 2.4.1 建立GOTO图

```

val setQueue: Deque<LRItemSet> = ArrayDeque(setOf(initSet))

```

```

initSet.id = setList.size
setList.add(initSet)

while (setQueue.isNotEmpty()) {
    val currSet: LRItemSet = setQueue.poll()
    val adjSet = adjacent[currSet.id] ?: mutableSetOf()

    for (sym in currSet.symbolsBehindDot) {
        val reachable = currSet.goto(sym)
        if (reachable !in setList) {
            reachable.id = setList.size
            setList.add(reachable)
            setQueue.add(reachable)
        }
        adjSet.add(DirectEdge(currSet.id, setList.indexOf(reachable),
sym))
    }

    adjacent[currSet.id] = adjSet
}

```

使用一个集合队列 `setQueue` 来保存所有的未处理LR项集，使用一个 `setList` 来保存所有产生的LR项集，对于一个LR项集，使用 `goto()` 方法产生所有能够到达的LR项集，如果新LR项集不存在 `setList` 中，则将其加入。并使用一个 `val adjacent: MutableMap<Int, MutableSet<DirectEdge>>` 邻接表来保存项集之间的有向边。

## 2.4.2 建立预测分析表

```

sealed class Action(val id: Int) {
    class Shift(id: Int): Action(id)
    class Reduce(id: Int): Action(id)
    class GOTO(id: Int): Action(id)
    class ACC: Action(0)
}

```

使用 `val table: MutableMap<Pair<Int, Symbol>, Action>` 来表示一个二维表，以一个集合标号和Symbol的二元组来作为二维表的索引，其中Action是一个密封类，来让我们较为容易地判断预测分析表中的具体动作。

```

    for (itemSet in graph.itemSets) {
        for ((sym, prodId) in itemSet.reduceAbleSet()) {
            val key = itemSet.id to sym
            if (prodId == 0) table[key] = Action.ACC()
            else table[key] = Action.Reduce(min(prodId, table[key]?.id ?:
Int.MAX_VALUE))
        }
    }
}

```

- 遍历GOTO图中所有的 LR项集
- 获得该项集中能够进行规约的二元组
- 如果是零号产生式，则向表中加入ACC，否则加入Reduce
- 在加入Reduce时，该表项可能会产生规约-规约冲突，此时选择产生式Id较小的那个，因为Id较小产生式比较靠前，优先级高

```

for ((from, to, sym) in graph.edges) {
    val key = from to sym
    when(sym) {
        is Symbol.TerminalSymbol -> {
            val isBeforeOp = key in table.keys &&
grammar.getOperator(table[key]!!.id) in OPERATORS.keys
            val isAfterOp = sym.content in OPERATORS.keys
            if (isBeforeOp && isAfterOp) {
                val firstOp = grammar.getOperator(table[key]!!.id)
                val secondOp = sym.content
                if (OPERATORS.getValue(secondOp) >
OPERATORS.getValue(firstOp)) {
                    table[key] = Action.Shift(to)
                }
            } else {
                table[key] = Action.Shift(to)
            }
        }
        is Symbol.NonterminalSymbol -> table[key] = Action.GOTO(to)
    }
}

```

- 对GOTO图的有向边进行遍历
- 如果是一个非终结符，向表中加入GOTO
- 如果是一个终结符，如果没有产生操作符移入-规约冲突，则向表中加入Shift
- 如果产生操作符移入-规约冲突，则通过判断两个操作符的优先级，来决定最后是移入还是规约

## 2.5 根据预测分析表进行规约(详见analyzeTable.kt)

```

while (!isAcc) {
    val s = stateStack.peek()

```

```

when(val action = table[s to Symbol.TerminalSymbol(currSym)]) {
    is Action.Shift -> {
        stateStack.push(action.id)
        // move to next symbol
        scanner += 1
        tempPointer += 1
        currSym = tokens[scanner]
    }
    is Action.Reduce -> {
        // A -> B
        val prod = grammar.prods[action.id]
        // pop num |B|
        res.add("sentence: " + tempTokens.joinToString(separator = " "))

        for (symbol in prod.derive) {
            tempPointer -= 1
            stateStack.pop()
            // replace E -> E * E with E
            tempTokens.removeAt(tempPointer)
        }
        tempTokens.add(tempPointer, prod.symbol.content)
        tempPointer += 1

        // t is stack peek, so push(GOTO[t, A])
        stateStack.push(table[stateStack.peek() to
prod.symbol]!!.id)
        res.add("reduce: " + prod.getHumanString())
        res.add("")
    }
    is Action.ACC -> {
        res.add("result: " + tempTokens.joinToString(separator = " "))

        isAcc = true
    }
    null -> {
        val reduced =
            if(res.isNotEmpty())
                "Part of input have been reduced\n" +
res.joinToString("\n") + "\n"
            else "\n"
        val pointer = " ".repeat(tokens.take(scanner).sumBy {
it.length + 1 }) + "^"
        val error = ""Reduce broken at:
            |   ${tokens.joinToString(" ")}
            |   $pointer
            |   please check your input!!!
            |
        """.trimMargin()
    }
}

```

```

        throw ReduceException(reduced + error)
    }
}
}

```

在规约的过程中，对规约产生式进行记录，并记录sentence变化的过程。

使用when语句来对查表得到的Action进行子类判断

- 移入：push当前状态入栈
- 规约：pop状态出栈，并将sentence中的字符串替换为产生式的左部
- ACC：成功，跳出循环
- null：查表失败，抛出异常

## 3 Error handling

### 3.1 规约Token序列

如果通过预测分析表无法找到action时，抛出规约异常么，并提供用户友好的指示。

例如，对于一个赋值语句 `int a = 5;`，如果忘记输入赋值符号 `=`，则会提醒用户输入错误

```

/Library/Java/JavaVirtualMachines/jdk1.8.0_192.jdk/Contents/Home/bin/java ...
Exception in thread "main" yacc.exception.ReduceException:
Reduce broken at:
    'primitive_type' 'id' 'number' 'semicolon' $
                ^
please check your input!!!

at yacc.AnalyzeTable.reduce(AnalyzeTable.kt:142)
at yacc.LRParser.reduce(LRParser.kt:23)
at yacc.MainKt.main(Main.kt:9)
at yacc.MainKt.main(Main.kt)

Process finished with exit code 1

```

## 4 感受和实验评价

在本次编译原理实验中，我使用kotlin语言实现了一个yacc。通过本次实验，我更加深入理解了从上下文无关文法到构造LR预测分析表的过程。对于first(), closure(), goto()等主要方法有了更好地掌握。