2016 级《编译原理课程设计》总结报告（12 组）

2019 年 5 月 23 日

姓名	性别	班级	学号	所占比例	个人成绩
陆子旭	男	04	51160420	100%	

任务分工：（请用小四号宋体填写）

陆子旭：
1. iOS 界面；
2. 测试；
3. 词法分析；
4. 语法分析；

成绩评定：

词法分析		自底向上语法分析		中间代码生成	
自顶向下语法分析		语义分析		目标代码生成	
团队成绩			教师签章		
备注					

填写说明：

1、请将首页红色部分信息填全，其中：班级为 2 位数字，保留首位的 0；学号为 8 位数字，软件学院以 54 开头；所占比例为百分数，精确到个位数，且所有人的所占比例之和为 100%；不足 3 人的分组请保留后面的多余空行，请勿修改该表的结构。
2、请根据实际情况填写任务分工部分，主要任务包括：编译系统的总体分析与设计，4 个具体功能的设计与实现，对应的测试与验证过程（报告正文需要列出若干组具体测试样例与对应结果），系统界面的设计与美工，以及辅助工具、视图和文件等。
3、成绩评定部分由指导教师填写，请勿填写和修改。

报告正文（请用小四号宋体填写，自行组织章节和段落）

一、 界面处理：

界面使用 swift 开发在 iOS 平台上，主要做了以下处理：

1. 适配了全部 iOS 设备，所有的 iOS 设备均可以有良好的运行界面；
2. 主体显示部分的任意拖放和不使用自动换行。因为在显示时有时需要显示大的表格，自动换行会带来显示上的不便；
3. 将默认的比例字体调节为等宽字体以便制表符的整齐输出；
4. 使用表格形式和树形形式的 String 形象输出；
5. 控制 TextView 的可编辑性，来准许输入代码；
6. 控制按钮的可交互性，来显示处理数据的依赖性。

二、 词法分析：

(一) 程序设计语言中的单词分为下面几类：

1. 保留字：PROGRAM，PROCEDURE，TYPE，VAR，IF，THEN，ELSE，FI，WHILE，DO，ENDWH，BEGIN，END，READ，WRITE，ARRAY，OF，RECORD，RETURN，INTEGER，CHAR；
2. 标识符：ID；
3. 常数：如 INTC，CHARC
4. 运算符：算术运算
5. 界限符：如分号、逗号、括号、单引号
6. 异常出错：ERROR（无法识别成上面几类，标记为错误，便于后续语法分析处理）
7. 结束符：ENDFILE。（留待用）

(二) TOKEN 结构：

TOKEN	column	int	记录该单词在源程序中的行数
	Line	int	记录该单词在源程序中的行数
	type	Terminal Type	记录该单词的词法信息， 其中 TerminalType 为单词的类型枚举
	data	String	记录该单词的语义信息

表 1 SNL 的 Token 结构内部表示

本编译器将词法分析和语法分析分别进行。主程序将首先调用词法分析程序，通过扫描源程序字符序列，按语言的词法规则将一个一个的字符组合成各类单词符号，并填写 TOKEN 信息，然后将结果输出到图形化界面上，同时把结果传给语法分析部分以便读取具有独立意义的单词符号。

(三) 基本实现：

我使用了有限状态自动机(DFA)来识别程序设计语言的单词的二元运算符部分，用正则表达式处理其他部分。即：

1. 先去掉注释；
2. 扫描得出符号，二元运算符用 switch-case 写出自动机，一元直接匹配，通过符号分割字符；

3. 找保留字；
4. 正则表达式判断其余类型。

(四) 注意点：

1. 保留字和标识符名字的区分：由于保留字的词法规则符合标识符的词法规则，故在处理时应加以区分。当扫描完一个单词时，先调用函数查保留字表，以确定是否是保留字，如果是则将类型标记为该单词在保留字表中对应的项。接着在扫描过程中只要遇到符合标识符词法规则的单词，如果没有出错，则统一标记为标识符三种中的一种；
2. 二元单词的处理：在程序设计语言中，有一类单词是由两个或两个以上符号组成，这类单词的前部分也可以是一个独立的单词，如:=等，在处理到“:”时，我还不能断定这个单词是“:”还是“:=”，这取决于“:”的下一个字符，如果下一个字符是“=”，则为“:=”，否则该单词为“:”。在处理这类单词时，是无法简单使用正则表达式的，要加以注意；
3. 注释的处理：在SNL 源程序语言中，符号“{”用作注释的开始符号，注释以符号 “}”的第一次出现作为结束。下面是一个注释的例子：{ this is a single {comment }。在第一个“{”和第一个“}”之间的部分均为注释部分，词法分析程序将注释中的内容略过不输出。

(五) 基本代码：

1. Token 结构：

```
struct Token{
    var type: TerminalType
    var data: String
    var line: Int
    var column: Int
}
```

2. 保留字查找：

```
let reservedWord:[String:TerminalType] = [
    "program":  .PROGRAM,
    "procedure":.PROCEDURE,
    "type":     .TYPE,
    "var":      .VAR,
    "if":       .IF,
    "then":     .THEN,
    "else":     .ELSE,
    "fi":       .FI,
    "while":    .WHILE,
    "do":       .DO,
    "endwh":    .ENDWH,
    "begin":    .BEGIN,
    "end":      .END,
    "read":     .READ,
    "write":    .WRITE,
```

```
        "array":    .ARRAY,
        "of":       .OF,
        "record":   .RECORD,
        "return":   .RETURN,
        "integer":  .INTEGER,
        "char":     .CHAR
    ]
```

3. 符号查找：

```
    let symbolWord:[Character:TerminalType] = [
        "=":.EQ,
        "<":.LT,
        "+":.PLUS,
        "-":.MINUS,
        "*":.TIMES,
        "/":.OVER,
        "(":.LPAREN,
        ")":.RPAREN,
        ".":.DOT,
        ":":.COLON,
        ";":.SEMI,
        ",":.COMMA,
        "[":.LMIDPAREN,
        "]":.RMIDPAREN,
        //    ":=":TerminalType.ASSIGN,
        //    "..":TerminalType.UNDERANGE
    ]
```

4. 类型判别：

```
    let discriminateType:[String:TerminalType] = [
        "\\d+":          .INTC,
        "\'.{1}\'":      .CHARC,
        "[a-zA-Z_][a-zA-Z_0-9]*":.ID
    ]
```

三、 语法分析：

(一) 语法分析的任务：根据语言的语法规则，对源程序进行语法检查，并识别出相应的语法成分。按照 SNL 编译程序的模型，语法分析的输入是从词法分析器输出的源程序的 Token 序列形式，然后根据语言的文法规则进行分析处理，语法分析的输出是无语法错误的语法成分，表示成语法树的形式。（如图一）

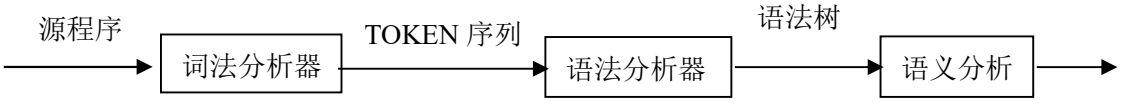

图 1 语法分析器的功能

(二) 重要集合（First 集，Follow 集，Predict 集）说明：在语法分析过程中，为了简化分析过程，提高分析效率，一般都对 2 型文法做一些限制，即要求文法满足一定的条件，不同的语法分析方法对语法的限制是不同的，受限后的语法是 2 型文法的子集，但它们一般都能描述程序设计语言的文法，这对于处理程序设计语言来说是足够用的。由于对文法做了适当的限制，从而提高了语法分析的效率（如避免了回溯问题），这样在语法分析中就可以利用一些特殊的条件进行特殊的处理，通常在语法分析中，会用到以下三个集合：First 集，Follow 集，Predict 集。First(β)表示 β 串所能推导的所有可能的终极符串的头终极符集，如果能推导出空串，则令 First(β)包含 ε。Follow(A) 表示所有那些终极符的集合，这些终极符在某个句型（句型是指由文法开始符推导出来的符号串，可以包含终极符和非终极符）中出现在 A 的紧后面，这种集合主要用于求 Predict（A→β）集合。集合相关代码如下：

1. 产生式：

```
struct Production {
    var productionLeft : NonTerminalType
    var productionRight = [String]()
}
```

2. 集合：

```
    var firstSet = [NonTerminalType : Set<String>]()
    var followSet = [NonTerminalType : Set<String>]()
    var predictSet = [Int : Set<String>]()
```

(三) 语法分析程序的输入输出：

1. 输入：利用词法分析产生的 Token 序列，每次从 Token 序列中取出一个 Token，作为程序正在分析的当前单词，其中存有其词法信息 Token 码和语义信息 Token 符号名，作为语法分析的输入。
2. 输出：SNL语法分析程序输出与源程序结构相对应的LL（1）语法分析树。

(四) LL（1）矩阵：可以运用 LL（1）矩阵来通过进入的 Token 的终极符，和文法上的非终极符来找到相应的产生式，相关代码如下：

```
func showTable() -> [[Int]] {
        var result : [[Int]] = [[Int]](repeating: [Int](repeating:
-1, count: TerminalType.allCases.count), count:
NonTerminalType.allCases.count)
        for (i, set) in predictSet {
            for p in set {
                result[NonTerminalType.allCases.firstIndex(of:
productionList[i].productionLeft)!][TerminalType.allCases.firstIn
dex(of: TerminalType.init(rawValue: p)!)!] = i
            }
        }
        return result
```

```
    }
```

(五) LL（1）语法树定义：在语法树节点中，需要链接父子节点、需要表示出终极符或者非终极符的类型，也需要存储 data 值。故语法树节点定义如下：

```
class Node {
    var parentNode : Node?
    var children = [Node]()
    var tType : TerminalType?
    var nType : NonTerminalType?
    var data = ""
 }
```

(六) LL（1）语法分析程序：自顶向下的 LL（1）分析程序需要维护两个表，一是读取的 Token 表，二是产生式的对应。在这里，我分了两个步骤：一是通过读取的 Token 来给产生式进行读取排序，最终会得到一个符合 Token 序列的自上而下的产生式序列。接着通过产生式序列深度遍历的方式来构造语法分析树，同时用一个总标记来标志 Token 序列的偏移，来记录读取 Token 序列中带有的数据。代码如下：

```
// 产生式的排序
    func productionListAnalyze() {
        var S = [String]()
        if let start = ProductionList.first {
            S.append(start.productionLeft.rawValue)
            var index = 0
            while !S.isEmpty {
                let s = S.last!
                if index < Tokens.count {
                    if let nt = NonTerminalType.init(rawValue: s) {
                        let                  i                  =
LLTable[NonTerminalType.allCases.firstIndex(of:
nt)!][TerminalType.allCases.firstIndex(of: Tokens[index].type)!]
                        if i > -1 {
                            Ps.append(ProductionList[i])
                            S.removeLast(1)
                            S                                    +=
ProductionList[i].productionRight.reversed()
                        } else {
                            print("ERROR: Production")
                            exit(1)
                        }
                    } else if let tt = TerminalType.init(rawValue:
s) {
                        if Tokens[index].type == tt {
                            S.removeLast(1)
                            index += 1
```

```
                        } else {
                            print("ERROR: NOT Match")
                            exit(-1)
                        }
                    } else {
                        S.removeLast(1)
                    }
                } else {
                    print("ERROR: Tokens")
                    exit(-1)
                }
            }
        }
    }
    //深度遍历构成树
    func dfsBuildTree(node : Node) {
        for i in Ps[PsIndex].productionRight {
            if let t = TerminalType.init(rawValue: i) {
                node.children.append(Node.init(parentNode:  node,
children: [Node](), tType: t, nType: nil, data: ""))
            } else if let t = NonTerminalType.init(rawValue: i) {
                node.children.append(Node.init(parentNode:  node,
children: [Node](), tType: nil, nType: t, data: ""))
            } else if i == null{
            } else {
                print("ERROR: Production")
                exit(-1)
            }
        }
        PsIndex += 1
        for i in node.children {
            if let _ = i.tType {
                i.data = Tokens[TokensIndex].data
                TokensIndex += 1
            } else {
                dfsBuildTree(node: i)
            }
        }
    }
```

(七)　树形展示：我需要将最终树的形状用 String 显示出来，需要生成一个能形象显示树的 String。代码如下：

```
    func showNode(lftstr : String, append : String, node : Node) ->
String {
```

```
            var b = append
            if let t = node.tType {
                b += t.rawValue
            } else if let t = node.nType {
                b += t.rawValue
            }
            if node.data != "" {
                b += "(" + node.data + ")"
            }
            b += "\n"

            if node.children.count > 0 {
                for (i, child) in node.children.enumerated() {
                    if i == node.children.count - 1 {
                        b += lftstr + showNode(lftstr: lftstr, append:
"|-", node: child)
                    } else {
                        b += lftstr + showNode(lftstr: lftstr + "| ",
append: "|-", node: child)
                    }
                }
            }

            return b
        }
```

结论（请用小四号宋体填写）

通过 4 周的实验，我实现了词法分析、LL（1）语法分析的基本功能，并且通过调试可以使得程序能正确运行并产生预期的结果。

按照书中介绍的方法和过程，参考已有的 SNL 编译程序的源代码，自己设计并实现词法分析和语法分析模块。整个编程过程锻炼了我的学习和调试程序的能力，同时对于一些不完善的细节通过我自己的努力进行重新编程和纠错，最终完成了程序的相关功能。这个过程也使得我对高级语言编程更加熟练，通过实际设计和开发，进一步地了解和掌握“编译程序的设计方法和实现技术”，也培养了我设计与开发大型软件的能力。

由于时间和个人能力有限，我的源程序还有一些细节不够完善，不过通过这次实验确实让我学到很多，我也会借此继续完善和强化自己，争取能不断地提高自己编程和实践能力，为以后的研究生学习和工作打下基础。