杭州电子科技大学

《编译原理课程实践》

实验报告

题　　目：语法分析算法实现

学　　院：计算机学院

专　　业：计算机科学与技术

班　级：23052322

学　号：23051214

姓　名：张逸轩

完成日期：2024.11.25

一、 实验目的

实现语法分析算法，深入理解算法本质。

二、 实验内容与实验要求

语法分析算法实现：

（1）实验任务 3.1 文法左递归消除算法实现；

（2）实验任务 3.2 文法左公因子提取算法实现；

（3）实验任务 3.3 FIRST 集合与 FOLLOW 集合计算算法实现；

（4）实验任务 3.4 LL（1）文法判定与预测分析器实现（选做）

三、 设计方案与算法描述

1.文法左递归消除算法

循环每个 I 前面的所有 j

将所有形如 Ai→Ajγ 的产生式替换为 Ai→δ1γ|δ2γ|…|δkγ，其中 Aj→δ1|δ2|…|δk 是 Aj 的当前产生式。

最后消除他们的直接左递归

2.文法左公共因子提取

首先建一个 字典树 Trie，实现可以提取最长公共前缀。

然后遍历每个非终结符的候选式，将它们插入 Trie

然后在 Trie 查找每个非终结符候选式的公共前缀。

最后生成新的式子。

3. First 和 Follow 集

First 集：先初始化，然后遍历所有符号，如果是终结符就是自己，否则就递归去 B1 B2… 去找（这里有记忆化），如果没有 epsilon 就结束，否则最后添加进去即可

Follow 集：先加 $ 到 开始符号，初始化。然后反复迭代，按照两个条件添加 FOLLOW, 只要有变化我们就继续迭代。最后得到 FOLLOW

4. LL(1)文法判定与预测分析器设计

LL（1）判断：

先生成 SELECT，然后判断两两有无交集，没有说明是。

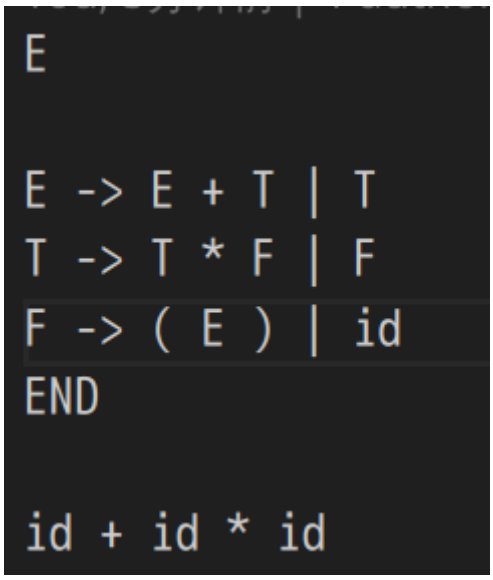
构造预测分析表：

把生成的 SELECT 直接构造成表

预测分析器:

建立栈，每次提取栈顶，如果是终结符号看匹不匹配，否则去找到预测表中对应的产生式，将产生式的符号逆序入栈。中间匹配不上就说明错误。

四、 测试结果



```
> python3 main.py < tests/test1
请输入文法开始符号: 已读取开始符号: 'E'
请输入文法（使用 'END' 来结束输入）:
已读取产生式: E -> E + T | T
已读取产生式: T -> T * F | F
已读取产生式: F -> ( E ) | id

原始:
开始符号: 'E'
非终结符号集: [E, T, F]
终结符号集: [+, id, *, ), (]
产生式:
E -> E + T | T
T -> T * F | F
F -> ( E ) | id

处理后:
开始符号: 'E'
非终结符号集: [E, T, F, E', T']
终结符号集: [+, id, *, ϵ, ), (]
产生式:
E -> T E'
T -> F T'
F -> ( E ) | id
E' -> + T E' | ϵ
T' -> * F T' | ϵ
```

```
FIRST 集:
FIRST(E) = {id, (}
FIRST(T) = {id, (}
FIRST(F) = {id, (}
FIRST(E') = {ϵ, +}
FIRST(T') = {ϵ, *}
FOLLOW 集:
FOLLOW(E) = {$, )}
FOLLOW(T) = {+, $, )}
FOLLOW(F) = {+, $, *, )}
FOLLOW(E') = {$, )}
FOLLOW(T') = {+, $, )}
满足LL(1): True
输入待分析串:输入待分析串:已获取输入串: 'id + id * id'
初始分析栈: ['$', 'E']
分析栈: ['T', "E'", '$'], 输入串: 'id + id * id $', 动作: E -> T E'
分析栈: ['F', "T'", "E'", '$'], 输入串: 'id + id * id $', 动作: T -> F T'
分析栈: ['id', "T'", "E'", '$'], 输入串: 'id + id * id $', 动作: F -> id
分析栈: ["T'", "E'", '$'], 输入串: '+ id * id $', 动作: match: 'id'
分析栈: ["E'", '$'], 输入串: '+ id * id $', 动作: T' -> ϵ
分析栈: ['+', 'T', "E'", '$'], 输入串: '+ id * id $', 动作: E' -> + T E'
分析栈: ['T', "E'", '$'], 输入串: 'id * id $', 动作: match: '+'
分析栈: ['F', "T'", "E'", '$'], 输入串: 'id * id $', 动作: T -> F T'
分析栈: ['id', "T'", "E'", '$'], 输入串: 'id * id $', 动作: F -> id
分析栈: ["T'", "E'", '$'], 输入串: '* id $', 动作: match: 'id'
分析栈: ['*', 'F', "T'", "E'", '$'], 输入串: '* id $', 动作: T' -> * F T'
分析栈: ['F', "T'", "E'", '$'], 输入串: 'id $', 动作: match: '*'
分析栈: ['id', "T'", "E'", '$'], 输入串: 'id $', 动作: F -> id
分析栈: ["T'", "E'", '$'], 输入串: '$', 动作: match: 'id'
分析栈: ["E'", '$'], 输入串: '$', 动作: T' -> ϵ
分析栈: ['$'], 输入串: '$', 动作: E' -> ϵ
分析栈: [], 输入串: '', 动作: match: '$'
分析结果: True
```

五、 源代码

```python
class CFG:
    def __init__(self, read=False):
        self.terminalSyms: set[str] = set()  # 终结符号集
        self.startSym: str = None  # 开始符号
        self.grammar: dict[str, list[list[str]]] = {}  # 产生式
        self.firstSets: dict[str, set[str]] = {}  # FIRST 集
        self.followSets: dict[str, set[str]] = {}  # FOLLOW 集
        self.predictiveTable: dict[str, dict[str, list[list[str]]]] = {}  # 预测分析表

        if read:
            self.read_grammar()
```

```python
def eliminate_left_recursion(self) -> None:
        """消除左递归的函数"""
        nonterminalSyms = list(self.grammar.keys())

        # 按顺序处理每个非终结符号
        for i in range(len(nonterminalSyms)):
            nonterminalSym = nonterminalSyms[i]
            productions = self.grammar[nonterminalSym]

            nonrecursiveProductions: list[list[str]] = []
            recursiveProductions: list[list[str]] = []

            # 消除间接左递归，遍历在当前非终结符号之前的所有非终结符号
            for j in range(i):
                # 对当前非终结符号的每个产生式进行检查
                for prod in productions.copy():
                    # Ai -> Ajβ
                    if prod[0] == nonterminalSyms[j]:
                        productions.remove(prod)
                        productions.extend(
                            [
                                prodj + prod[1:]
                                for prodj in self.grammar[nonterminalSyms[j]]
                            ]
                        )

            for newProd in productions:
                if newProd[0] == nonterminalSym:
                    recursiveProductions.append(newProd[1:])
                else:
                    nonrecursiveProductions.append(newProd)

            # 处理直接左递归
            if recursiveProductions:
                newNonterminalSym = f"{nonterminalSym}'"
                for prod in nonrecursiveProductions:
                    prod.append(newNonterminalSym)
                self.grammar[nonterminalSym] = nonrecursiveProductions
                for prod in recursiveProductions:
                    prod.append(newNonterminalSym)
                recursiveProductions.append(["ϵ"])
                self.add_rule(newNonterminalSym, recursiveProductions)
            else:
                self.grammar[nonterminalSym] = nonrecursiveProductions

```

```python
def extract_left_common_factors(self):
        """提取左公因式"""
        newGrammar: dict[str, list[list[str]]] = {key: [] for key in self.grammar}

        for nonterminalSym, productions in self.grammar.items():
            # 如果产生式右部符号个数≤1，直接赋值
            if len(productions) <= 1:
                newGrammar[nonterminalSym] = productions
                continue

            # 将产生式加入到 Trie 中
            trie = Trie(nonterminalSym)
            for prod in productions:
                trie.insert(prod)

            # 获取最长公共因子式
            commonPrefixes = trie.get_prefixes()

            if commonPrefixes:
                newNonterminalSym = nonterminalSym
                prefixMap: list[tuple[list[str], str]] = []
                # 根据前缀构造新规则
                for commonPrefix in commonPrefixes:
                    newNonterminalSym = f"{newNonterminalSym}'"
                    prefixMap.append((commonPrefix, newNonterminalSym))
                    newGrammar[newNonterminalSym] = []
                    newGrammar[nonterminalSym].append(
                        commonPrefix + [newNonterminalSym]
                    )

                # 分配规则到新非终结符
                for prod in productions:
                    for commonPrefix, newNonterminalSym in prefixMap:
                        commonPrefixLen = len(commonPrefix)
                        if prod[:commonPrefixLen] == commonPrefix:
                            newGrammar[newNonterminalSym].append(prod[commonPrefixLen:])
                            break
                    else:
                        newGrammar[nonterminalSym].append(prod)
            else:
                newGrammar[nonterminalSym] = productions

        self.grammar = newGrammar
```

计算 FIRST FOLLOW

```python
#递归计算FIRST集
    def compute_first(self, symbol: str) -> set[str]:
        if not self.firstSets:
            self.compute_firstSets()

        # 如果是终结符号，直接返回
        if symbol not in self.firstSets:
            return {symbol}

        # 如果已经计算过，直接返回
        if self.firstSets[symbol]:
            return self.firstSets[symbol]

        # 计算 FIRST 集
        for prod in self.grammar[symbol]:
            count = 0
            for prodSym in prod:
                symFirst = self.compute_first(prodSym)
                if "ϵ" not in symFirst:
                    self.firstSets[symbol].update(symFirst)
                    break
                count += 1
                self.firstSets[symbol].update(symFirst - {"ϵ"})
            if count == len(prod):
                self.firstSets[symbol].add("ϵ")

        return self.firstSets[symbol]
```

.

```python
def compute_firstSets(self) -> dict[str, set[str]]:
        """计算 FIRST 集"""
        if self.firstSets:
            return self.firstSets

        # 初始化
        for nonterminalSym in self.grammar.keys():
            self.firstSets[nonterminalSym] = set()

        # 遍历每个非终结 计算 FIRST 集
        for nonterminalSym in self.firstSets.keys():
            self.compute_first(nonterminalSym)

        return self.firstSets
```

.

```python
def compute_follow(self, symbol: str) -> set[str]:
        if not self.followSets:
            self.compute_followSets()
        return self.followSets[symbol]

    def compute_followSets(self) -> dict[str, set[str]]:
        """计算 FOLLOW 集"""
        if self.followSets:
            return self.followSets

        # 初始化所有非终结符的 FOLLOW 集
        for nonterminal in self.grammar.keys():
            self.followSets[nonterminal] = set()
        self.followSets[self.startSym].add("$")  # $ 为输入的结束符

        # 迭代直到所有 FOLLOW 集不再变化
        changed = True
        while changed:
            changed = False
            for nonterminal, productions in self.grammar.items():
                for production in productions:
                    for i, symbol in enumerate(production):
                        if symbol not in self.terminalSyms:  # 当前符号是非终结符
                            originalSize = len(self.followSets[symbol])

                            if i + 1 < len(production):  # 右侧还有符号
                                firstSet = self.compute_first_of_production(
                                    production[i + 1 :]
                                )
                                self.followSets[symbol].update(firstSet - {"ϵ"})
                                if "ϵ" in firstSet:
                                    self.followSets[symbol].update(
                                        self.followSets[nonterminal]
                                    )
                            else:  # 如果是最后一个符号，添加非终结符的 FOLLOW 集
                                self.followSets[symbol].update(
                                    self.followSets[nonterminal]
                                )

                            changed |= originalSize != len(self.followSets[symbol])

        return self.followSets
```

SELECT

```python
def compute_select_of_production(
        self, nonterminalSym: str, production: list[str]
    ) -> set[str]:
        """计算某个产生式的 SELECT 集"""
        selectSet: set[str] = set()

        # 计算产生式右部的 FIRST 集
        firstSet = self.compute_first_of_production(production)

        # 将 FIRST 集中除 ϵ 之外的符号加入 SELECT 集
        selectSet.update(firstSet - {"ϵ"})

        # 如果在里面，再加个 FOLLOW（A）
        if "ϵ" in firstSet:
            selectSet.update(self.compute_follow(nonterminalSym))

        return selectSet
```

LL(1)

```python
def construct_predictive_table(self) -> dict[str, dict[str, list[list[str]]]]:
        """构造 LL(1) 预测分析表"""
        self.predictiveTable = {
            nonterminal: {terminal: [] for terminal in self.terminalSyms | {"$"}}
            for nonterminal in self.grammar.keys()
        }

        #遍历所有非终结符及其产生式
        for nonterminal, productions in self.grammar.items():
            for prod in productions:
                selectSet = self.compute_select_of_production(nonterminal, prod)

                # 给 每个 SELECT 填进去
                for terminal in selectSet:
                    self.predictiveTable[nonterminal][terminal].append(prod)

        return self.predictiveTable
```

预测过程

```python
def parse(self, inputStr: list[str]) -> bool:
        if not self.predictiveTable:
            self.construct_predictive_table()
        if not self.is_ll1():
            print("\033[31m该文法不是LL(1)文法\033[0m")
            return False


        stack: list[str] = ["$", self.startSym]
        print("初始分析栈:", stack)

        inputStr.append("$")
        while stack:
            top = stack.pop()
            curSym = inputStr[0]
            action: str = None

            # 判断栈顶是终结符号
            if top in self.terminalSyms | {"$"}:
                if top == curSym:
                    # 匹配
                    action = f"match: '{curSym}'"
                    inputStr = inputStr[1:]
                else:
                    return False
            else:
                # 预测分析表中找到对应的产生式
                productions = self.predictiveTable[top][curSym]
                if productions:
                    production = productions[0]
                    # 将产生式的符号逆序入栈
                    for sym in reversed(production):
                        if sym != "ε":
                            stack.append(sym)
                    action = f"{top} -> {" ".join(production)}"
                else:
                    return False

            print(f"分析栈: {list(reversed(stack))}, 输入串: '{" ".join(inputStr)}', 动作: {action}")

        return True
```