Decaf PA1_B 实验报告

计62 李祥凡 2016011262

第一部分 具体实现

1. 词法分析

直接将PA1_A中由Lexer.l生成的Lexer.java复制过来即可

2. 语法分析

跟PA1_A的大致做法类似

1）在Tree.java中添加必要的语法节点类(直接从PA_1A中拷贝新定义的类过来即可)

2）在Parser.spec中定义新的语法规则，这里要将PA_1A中定义的文法通过消除左递归、提取左公因式等方法改变为等价的LL(1)文法。在操作符的优先级以及结合性方面，本次PA要求自己实现，实现方法：

优先级：

以 “%%” 操作符为例，它的运算优先级低于 '+' 和 '-', 定义新的表达式 ExprArrayRepeat, 使其作为 以 '+' 或 '-' 为顶层操作符的表达式 Expr5 的父表达式：

```
585 ExprArrayRepeat :   Expr5 ExprArrayRepeatT
586                     {
587                         $$.expr = $1.expr;
588                         if ($2.svec != null) {
589                             for (int i = 0; i < $2.svec.size(); ++i) {
590                                 $$.expr = new Tree.Binary($2.svec.get(i), $$.expr,
591                                     $2.evec.get(i), $2.lvec.get(i));
592                             }
593                         }
594                     }
595                 ;
596
597 ExprArrayRepeatT:   ArrayRepeat Expr5 ExprArrayRepeatT
598                     {
599                         $$.svec = new Vector<Integer>();
600                         $$.lvec = new Vector<Location>();
601                         $$.evec = new Vector<Expr>();
602                         $$.svec.add($1.counter);
603                         $$.lvec.add($1.loc);
604                         $$.evec.add($2.expr);
605                         if ($3.svec != null) {
606                             $$.svec.addAll($3.svec);
607                             $$.lvec.addAll($3.lvec);
608                             $$.evec.addAll($3.evec);
609                         }
610                     }
611                 |   /* empty */
```

结合性：

以 “++” 操作符为例，只需在ExprArrayConcat对应的Action中改变生成Binary节点类的顺序即可, "++"为右结合, 则Binary节点由右向左生成：

```
552 ExprArrayConcat :   ExprArrayRepeat ExprArrayConcatT
553                     {
554                         if ($2.svec != null) {
555                             $$.expr = $2.evec.get($2.evec.size()-1);
556                             for (int i = $2.svec.size()-2; i >= 0 ; --i) {
557                                 $$.expr = new Tree.Binary($2.svec.get(i+1), $2.evec.get(i), $$.expr,
558                                     $2.lvec.get(i+1));
559                             }
560                             $$.expr = new Tree.Binary($2.svec.get(0), $1.expr,
561                                     $$.expr, $2.lvec.get(0));
562                         }
563                         else
564                             $$.expr = $1.expr;
565                     }
566                 ;
```

3）在SemValue类中添加必要的成员变量

3. 增加错误恢复功能

实现了README中描述的方法，修改Parser.java类中的parse函数，最初follow集合 (就是parse函数的第二个参数) 是没有用到的，始终是个空集合，现在要用follow集合来表示当前节点的所有父节点的followSet集合的并集：在parse函数的开始记录下一个follow集合的拷贝，然后将follow集合并上当前的symbol的followSet, 递归地向下parse，递归完所有子节点后，将follow集合恢复原状:

```
HashSet<Integer> ori_follow = new HashSet<Integer>();
for(Integer i : follow) {
    ori_follow.add(i);
}
follow.addAll(followSet(symbol));
for (int i = 0; i < length; i++) { // parse right-hand side symbols one by one
    int term = right.get(i);
    params[i + 1] = isNonTerminal(term)
            ? parse(term, follow) // for non terminals: recursively parse it
            : matchToken(term) // for terminals: match token
            ;
}

follow.clear();
for(Integer i : ori_follow) {
    follow.add(i);
}
```

对于非终结符A，当前输入符号为a, 若a∉ Begin(A), 则跳过所有不在Begin(A) ∪ End(A) 中的符号，若遇到的是Begin(A) 中的符号，则继续原有的语法分析，否则返回一个新的SemValue对象：

```
if(!(beginSet(symbol).contains(lookahead)))
{
    error();
    while((!(beginSet(symbol).contains(lookahead) || followSet(symbol).contains(lookahead) || follow.contains(lookahead)))
    {
        lookahead = lex();
    }
    if(!(beginSet(symbol).contains(lookahead)))
    {
        return new SemValue();
    }
}
```

第二部分： else为空冲突处理原理

给产生式依据先后顺序设立优先级，放在前面的产生式优先级比较高。发生冲突时优先使用优先级高的产生式。本例中将E else S的优先级设为高于E */empty/*，因此二者均可匹配时优先匹配非空的产生式，产生的结果也符 合C++、Java等传统编程语言的标准：else与上方最近的if块匹配。

示例代码：

```
class Main {
    static int main() {
        if(true)
            if(false)
                Print(1);
            else
                Print(2);
            return 0;
        }
    }
```

此例中，else语句块既可以与第一个if匹配——此时程序将不会执行Print语句，又可以与第二个if语句块匹配——此时将会执行 Print(2) 语句。 由于优先匹配规则，先和最近的if匹配，因此最终会执行 Print(2)。这段代码的执行逻辑可以由其产生的语法树清晰地看出来：

```
program
    class Main <empty>
        static func main inttype
            formals
            stmtblock
                if
                    boolconst true
                    if
                        boolconst false
                        print
                            intconst 1
                    else
                        print
                            intconst 2
                return
                    intconst 0
```

由缩进情况可知，else和最近的(第二个)if为一个整体。

第三部分：为什么原先的comprehension表达式文法改成LL(1) 比较困难

按原先方法写产生的冲突：

```
[java] Warning: conflict productions at line 937:
[java] Constant -> LITERAL
[java] Constant -> NULL
[java] Constant -> ArrayConst
[java] Constant -> '[' CompArrayStmt ']'
[java] Warning: unreachable production:
[java] Constant -> '[' CompArrayStmt ']'
[java] predictive set is empty
```

由冲突信息可以看出，该文法会和数组常量表达式冲突，然而，这二者很难提取公因子，因为二者做的操作完全不一样：一个是执行首个表达式的内容并将结果作为表达式的值；一个是将首个表达式和后续的表达式拼接成为一个数组。这会给语义分析带来困难。

第四部分：误报举例

如下所示的错误代码，method函数体缺少了左尖括号 '{':

```
class Father {
    int field;

    void method(int f)
        if (f > 0 && f < 10) {
            f = f * 3;
            return method(f);
        }
    }
}
```

产生的报错信息:

```
*** Error at (5,9): syntax error
*** Error at (9,5): syntax error
```

即不仅缺失的左尖括号位置报了错，应当与缺失的左尖括号匹配的右尖括号位置也报了错，而这个报错是多余的，即缺失的左尖括号并没有被修复 (补全) ，显然属于误报。

实验总结

我花了大概一天半的时间完成了本次PA, 我认为本次实验的难点在于使新定义的语法规则满足LL(1)文法 ，增加新的语言特性的同时满足特定的优先级和结合律。在原有架构的基础上完成错误恢复的实现其实较为简单，但我最初对实验要求的理解出了一些偏差，README中有一句话“在这一阶段，你无需关心第二个参数follow”，我以为“这一阶段”指的是整个实验阶段.....，于是浪费了不少时间。