PA1-B 实验报告

任务一、二：添加抽象类和局部类型推断

由于 ABSTRACT 和 VAR 的添加与否不影响 LL(1)文法的规范性，因而可以直接移植 PA1-A 的代码（因为两个关键字都加在推导式的最前方，在读取 nextToken 的过程中不会引起左递归、判断等问题）。同时，沿用 PA1-A 的关键字注册、类声明与注册。

任务三：添加简单的错误恢复

1.实现思路：

按照实验指导书-实验内容一节的内容，采用如下错误恢复方法：当分析非终结符 A 时，若当前输入符号 a∉Begin(A)，则先报错，然后跳过符号串中的一些符号，直至遇到 Begin(A)∪End(A)中的符号：若遇到的是 Begin(A)中的符号，可恢复分析 A；若遇到的是 End(A)中的符号，则 A 分析失败，继续分析 A 后面的符号。若符号属于两者的交集，则也判断失败。

2.具体实现：

```
private SemValue parseSymbol(int symbol, Set<Integer> follow) {
    boolean hasError = false;
    var bSet = setCopy(beginSet(symbol));
    var eSet = setCopy(followSet(symbol));
    eSet.addAll(follow);
    //only when begin and end both don't contain the token, it goes wrong
    if(!bSet.contains(token)) {
        //print it's wrong
        hasError = true;
        yyerror("syntax error");
        //jump some tokens
        while(!bSet.contains(token) && !eSet.contains(token)){
            token = nextToken();
        }
        //after the circulation, the token right now belongs to bSet orelist or both
        //if it belongs to eSet, the analysis is a failure
        if(eSet.contains(token)){
            return null;
        }
        //else if it's only belongs to bSet, continue the analysis.
    }

    var result = query(symbol, token); // get production by lookahead symbol
    var actionId = result.getKey(); // get user-defined action

    var right = result.getValue(); // right-hand side of production
    var length = right.size();
    var params = new SemValue[length + 1];
```

```
for (var i = 0; i < length; i++) { // parse right-hand side symbols one by one
    var term = right.get(i);
    params[i + 1] = isNonTerminal(term)
            ? parseSymbol(term, eSet) // for non terminals: recursively parse it
            : matchToken(term) // for terminals: match token
    ;
    if(params[i+1] == null) {
        hasError = true;
    }
}

if(hasError){
    return null;
}

act(actionId, params); // do user-defined action
return params[0];
```

按照上述逻辑进行编写即可。式中提到的 setCopy 函数是对集合进行深拷贝的方法，具体实现如下：

```
private Set<Integer> setCopy(Set<Integer> s){
    var temp = new TreeSet<Integer>();
    temp.addAll(s);
    return temp;
}
```

任务四：添加函数类型、Lambda 表达式及函数调用

1°函数调用

函数调用的过程也可沿用 PA1-A 的方法，直接将当时更改过的 Call 类型的函数粘贴过来即可。然后在 Decaf.spec 文件中进行改动，一共有三处：AfterLParen 中 else if 一处，将 Id 删去；Expr8 中 else if 一处，将 Id 删去；Expr9 中，将最后一个产生式含有 Id 的一句删去。

```
AfterLParen     :   CLASS Id ')' Expr7
                    {
                        $$ = svExpr(new ClassCast($4.expr, $2.id, $4.pos));
                    }
                |   Expr ')' ExprT8
                    {
                        $$ = $1;
                        for (var sv : $3.thunkList) {
                            if (sv.expr != null) {
                                $$ = svExpr(new IndexSel($$.expr, sv.expr, sv.pos));
                            } else if (sv.exprList != null) {
                                $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
                            } else {
                                $$ = svExpr(new VarSel($$.expr, sv.id, sv.pos));
                            }
                        }
                        $$.pos = $$.expr.pos;
                    }
                ;
```

```
Expr8           :   Expr9 ExprT8
                    {
                        $$ = $1;
                        for (var sv : $2.thunkList) {
                            if (sv.expr != null) {
                                $$ = svExpr(new IndexSel($$.expr, sv.expr, sv.pos));
                            } else if (sv.exprList != null) {
                                $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
                            } else {
                                $$ = svExpr(new VarSel($$.expr, sv.id, sv.pos));
                            }
                        }
                        $$.pos = $$.expr.pos;
                    }
                ;
```

```
|   Id
    {
        $$ = svExpr(new VarSel($1.id, $1.pos));
    }
;
```

2°添加 Lambda 表达式：

观察 Op1~Op7 定义的各运算符可发现，Expr1~Expr9 是按照优先度从低到高的顺序排列的。在 Lambda 表达式中，优先度最低的是'=>'，因而创建了一个 Expr0 用来套入它。接下来分为后面接 block 和'=> expr'两种情况讨论。同时，为了避免左递归，进行了提取公因式的操作。

```
Expr            :   Expr0
                    {
                        $$ = $1;
                    }
                |   Expr1
                    {
                        $$ = $1;
                    }
                ;

Expr0           :   FUN '(' VarList ')' FuncBody
                    {
                        $$ = svExpr(new Lambda($1.pos, $3.varList, $5.expr, $5.block));
                    }
                ;

FuncBody        :   Block
                    {
                        $$ = $1;
                    }
                |   ARROW Expr
                    {
                        $$ = $2;
                    }
                ;
```

3°添加函数类型

函数类型

语法规范：

```
type ::= ...
       | type '(' typeList ')'

typeList ::= (type (',' type)*)?
```

括号左边的是返回值的类型，括号内的是诸参数的类型。

通过观察注意到 type 的定义与 AtomType('null')*类似，因此可以仿照后者与 ExprT 的构造方式进行构造。在第一个 AtomType 后利用一个函数遍历 ArrayType 的 thunklist，从而生成一个 type(,type)*的序列来。TypeList 的构造与 PA1-A 一致，进行 LL(1)文法改造之后如下图所示。如果 ArrayType 的当前位置的 typeList 不为空，则由此产生一个 Lambda 表达式，否则产生一个 TArray。

```
Type        :   AtomType ArrayType
                {
                    $$ = $1;
                    for (var sv: $2.thunkList) {
                        if(sv.typeList != null) {
                            $$.type = new TLambda($1.pos, $1.type, sv.typeList);
                        }
                        else {
                            $$.type = new TArray($$.type, $1.type.pos);
                        }
                    }
                }
            ;

ArrayType   :   '[' ']' ArrayType
                {
                    var sv = new SemValue();
                    $$ = $3;
                    $$.thunkList.add(0, sv);
                }
            |   '(' TypeList ')' ArrayType
                {
                    var sv = new SemValue();
                    sv.typeList = $2.typeList;
                    $$ = $4;
                    $$.thunkList.add(0, sv);
                }
            |   /* empty */
                {
                    $$ = new SemValue();
                    $$.thunkList = new ArrayList<>();
                }
            ;

TypeList    :   Type TypeList1
                {
                    $$ = $2;
                    $$.typeList.add(0, $1.type);
                }
            |   /* empty */
                {
                    $$ = svTypes();
                }
            ;
```

```
TypeList1   :   ',' Type TypeList1
                {
                    $$ = $3;
                    $$.typeList.add(0, $2.type);
                }
            |   /* empty */
                {
                    $$ = svTypes();
                }
            ;
```

实验中遇到的困难：

实验中最大的困难在于 LL(1)文法的相关改写，如同上面实验所述，TypeList 的构造其实早在 PA1-A 就

已经完成，但是当时的构造方式是由如下的上下文无关文法构造而来，存在左递归的问题，设计合适的 LL(1) 文法是一个很大的问题。

S -> (S) | A

A -> type | type ‘,’ A

其次，是在已有代码的框架中找寻类似结构的代码启示构造的过程，从 Op1~Op7 的定义找到优先度的关系再安排’=>’的位置，又或者是通过 ExprT 来构造 type 序列，这些都需要仔细阅读代码才能发现。

思考题：

1.本阶段框架是如何解决空悬 else（dangling-else）问题的?

答：若碰到 if if else 形式的代码块，由于 LL(1)文法每次只读取一个 Token，当读到第一个 if 的时候会进到 Stmt 下面的产生式 IF ‘(‘ Expr ‘)’ Stmt 中。此时在进行 ParseSymbol 分析的时候，会将第二个 if 继续识别成一个 Stmt 并送到外层 Stmt 的 if 类的构造函数中，换言之，第二个 if 将成为第一个 if 的附属。此时按照正常的语义，第一个 if 的 block 已经结束，从而 else 将会和第一个 if 关联起来。空悬 else 问题得到解决。

2.使用 LL(1)文法如何描述二元运算符的优先级与结合性？请结合框架中的文法，举例说明。

答：在本阶段的框架中，所有的运算符根据优先度从低到高的顺序分别写在了 Op1~Op7 中，同一个 Op 中的表达式优先度是一样的。从顶向下分析的过程是将匹配到的 Expr 进行拆分，一直拆分到最下层，再进行计算，每一层的计算由于优先度相同而可以进行结合，计算完成后返回到上一层的运算，这里体现出了优先级。对于左右结合，框架的一个做法是用 thunkList 控制循环是顺序还是倒序来控制是左结合还是右结合，比如 buildBinaryExpr 函数就是左结合。

3.无论何种错误分析方法，都无法完全避免误报的问题。请举出一个具体的 Decaf 程序（显然它要有语法错误），用你实现的错误恢复算法进行语法分析时会带来误报。并说明该算法为什么无法避免这种误报。

答：abstract1.decaf:

```
class Main {
    abstract int v;
    static void main() { }
}
```

文法分析程序运行到 abstract 的时候，会进入到 FieldList 下的 ABSTRACT Type Id ‘(‘ VarList ‘)’ ‘;’ FieldList 中。匹配完 Type 和 Id 之后，应该匹配’(‘，但是读到了’;’，第一次报错。’;’不属于 Begin 或 End 集合，于是被跳过。继续读，读到了 static，属于 End 集合，‘(’分析失败，进到 VarList 分析，可认为匹配到 /* empty*/。继续匹配’)’，此时 Token 还是 static，于是同之前一样报错，并且跳过。然后进到最后的 FieldList 匹配，正确结束，整个过程报错两次。然而按照正常的逻辑应该只报错一次，即 abstract int v 是正常的 int v 声明前多写了一个 abstract 而已。这个问题本身是不可复原的，也从来没有一个正确的标准。所谓的正常理解也只是另一个角度的观点而已，误报只是在乎是否合乎人们的正常逻辑，而已，因此无法避免。