编译原理 PA1-B 实验报告

计 71 张程远 2017011429

1 工作简述

本次实验任务可大体分为两项：一是将 PA1A 的工作整理到 PA1B 上，二是实现错误恢复的代码。

1.1 工作迁移

这个阶段最大的变化在于从 LR 文法到 LL(1)文法的转变。其余工作，例如在 Tree 中增加相应的解析函数，在 jflex 和 parser 中增加相应的 Token 等，与 PA1A 基本一致。

abstract 与 var

此处的文法添加与 PA1A 基本相同，完全按照文法要求即可。

```
ClassDef : ...
          | ABSTRACT CLASS Id ExtendsClause '{' FieldList '}'
            {
                $$ = svClass(new ClassDef(true, $3.id, Optional.ofNullable($4.id), $6.fieldList, $2.pos));
            }
MethodDef : ...
           | ABSTRACT Type Id '(' VarList ')' ';' FieldList
             {
                $$ = $8;
                $$.fieldList.add(0, new MethodDef($3.id, $2.type, $5.varList, $3.pos));
             }
SimpleStmt : ...
           | VAR Id '=' Expr
             {
                $$ = svStmt(new LocalVarDef($2.id, $3.pos, $4.expr, $2.pos));
             }
```

ClassDef、MethodDef 以及 LocalVarDef 类中提供了新的构造函数以适配新增的解析方式。

函数类型

这部分修改的难度较大。最开始我的想法是在 Type 后面直接消左递归，试了几次发现只能通过部分测例，经常会有 Nullpt 的 bug。我与其他同学交流后，发现实际上()与[]在地位上是完全平等的，因此只需要读懂多维数组的实现方法就可以了。

在 ArrayType 里，框架利用 thunklist 实现了[]嵌套的功能，其中用 intVal 来计数。我们考虑让[]与()等价，首先要在 ArrayType 里做这样的修改。

```
1.  ArrayType : '[' ']' ArrayType
2.              {
3.                 $$ = $3;
4.                 $$.intVal++;
5.                 var sv = new SemValue();
6.                 $$.thunkList.add(0, sv);
7.              }
8.            | '(' TypeList ')' ArrayType
9.              {
10.                $$ = $4;
11.                $$.intVal++;
12.                var sv = new SemValue();
13.                sv.typeList = $2.typeList;
14.                $$.thunkList.add(0, sv);
15.             }
```

这样 thunklist 的每一个元素要么是(TypeList)要么是[ ]，地位完全平等。接下来在 Type 里取出每个元素做处理。

```
1.  Type: AtomType ArrayType
2.       {
3.          $$ = $1;
```

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

最后给 empty 做处理，给它的 thunklist 初始化。

```
ArrayType: ...
         |   /* empty */
            {
               $$ = new SemValue();
               $$.intVal = 0; // counter
               $$.thunkList = new ArrayList<>();
             }
```

关于 TypeList 的处理，核心还是模仿 VarList 的写法。与 PA1A 相同，需要在 SemValue 里提供 typeList 和 svTypes()的定义。

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

```
            }
        ;

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

接下来的部分是与 PA1A 最大的不同之处：作为一种类型，函数类型还需要能够被 new 成数组。这个地方文法中没有明确提到，PA1A 又不需要做，公开测例又检测不出来，因此是非常坑的。最开始的想法是将 AfterNewExpr 的 AtomType 改为 Type，但这样会导致文法不符合 LL（1），而且 Type 实际上也过于强了，例如 Type [][3]，前面的空括号完全可以交给 Type 而不是 AfterNewExpr 来处理。所以这里考虑在[]中自由插入'(' TypeList ')'的形式，具体修改如下：

```
AfterNewExpr    :   Id '(' ')'
                    {
                        $$ = svId($1.id);
                    }
                |   AtomType T '[' AfterLBrack
                    {
                        $$ = $1;
                        for (var sv : $2.thunkList){
                            if(sv.typeList != null){
                                $$.type = new TLambda($1.type, sv.typeList, $1.type.pos);
                            }
                        }
                        for (var sv : $4.thunkList){
```

```
                            if(sv.typeList != null){
                                $$.type = new TLambda($1.type, sv.typeList, $1.type.pos);
                            }
                            else {$$.type = new TArray($1.type, $1.type.pos);}
                        }
                        $$.expr = $4.expr;
                    }
                ;
AfterLBrack     :   ']' T '[' AfterLBrack
                    {
                        $$ = $4;
                        $$.thunkList.addAll(0, $2.thunkList);
                        var sv = new SemValue();
                        $$.thunkList.add(0, sv);
                    }
                |   Expr ']'
                    {
                        $$ = svExpr($1.expr);
                        $$.thunkList = new ArrayList<>();
                    }
                ;
T               :   '(' TypeList ')' T
                    {
                        $$ = $4;
                        var sv = new SemValue();
                        sv.typeList = $2.typeList;
                        $$.thunkList.add(0, sv);
                    }
                |   /* empty */
                    {
                          $$ = new SemValue();
                          $$.intVal = 0; // counter
                          $$.thunkList = new ArrayList<>();
                    }
                ;
```

T 用于产生小括号。两个 thunklist 保存的内容需要依次取出。

Lambda 表达式

实验指导已经给了明确的文法定义：让它与 Expr1 并列。同时'=>'的优先级最低，因此就将其放到 Op0 的位置。这里需要提取左公因子，将提取后的部分称为 AfterFun。

```
Expr :   Expr1
         {
            $$ = $1;
         }
       |   FUN '(' VarList ')' AfterFun
         {
            if ($5.block != null){
               $$ = svExpr(new Lambda($3.varList, $5.block, $1.pos));
         }
            else{
               $$ = svExpr(new Lambda($3.varList, $5.expr, $1.pos));
            }
         }
     ;
AfterFun : Op0 Expr
            {
               $$ = $2;
            }
          |   Block
            {
               $$ = $1;
            }
          ;
```

Call

Call 的修改同样也比较困难。通过在 spec 里搜索可以知道有 3 处 call，分别代表 call 原本的几种文法。将 receiver 改成 expr 后，条件宽泛了许多，因此前两处提供了 expr 的 call 直接去掉 ID 参数并保留。最后一处没有 expr，直接去掉它的 call 操作。另外，ExprListopt 的存在会使得 LL1 文法出现冲突。因此我们需要去掉它，并在其原来起作用的地方换成等价

的、符合 LL1 文法的表达。具体而言，是修改 ExprT8，并删除 Expr9 最后部分的 ExprListOpt。

```
AfterLParen : ...
                } else if (sv.exprList != null) {
                      $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
                  }
Expr8       : ...
                } else if (sv.exprList != null) {
                      $$ = svExpr(new Call($$.expr, sv.exprList, sv.pos));
                  }
Expr9       : ...
            |    Id
                {
                      $$ = svExpr(new VarSel($1.id, $1.pos));
                 }
             ;
ExprT8 :   ...
       |   '.' Id ExprT8
          {
             var sv = new SemValue();
             sv.id = $2.id;
             sv.pos = $2.pos;
             $$ = $3;
             $$.thunkList.add(0, sv);
           }
       |   '(' ExprList ')' ExprT8
          {
             var sv = new SemValue();
             sv.pos = $1.pos;
             if ($2.exprList != null) {
```

```
                sv.exprList = $2.exprList;
                sv.pos = $1.pos;
             }
             $$ = $4;
             $$.thunkList.add(0, sv);
           }
```

1.2 错误恢复

直接按照算法来。beginA 和 endA 已经在生成的 LLTable.java 里有定义，直接拿来用，按照流程翻译过来就行。

```
private SemValue parseSymbol(int symbol, Set<Integer> follow) {
            var result = query(symbol, token); // get production by lookahead symbol
            Set<Integer> beginS = beginSet(symbol);
            Set<Integer> endS = followSet(symbol);
            endS.addAll(follow);
            if (!beginS.contains(token)) {
                haveError = true;
                yyerror("syntax error");
                while (true) {
                    if (beginS.contains(token)) {
                        result = query(symbol, token);
                        break;
                    }
                    else if (endS.contains(token)) {return null;}
                    token = nextToken();
                }
            }
            var actionId = result.getKey(); // get user-defined action
```

```
19.
20.             var right = result.getValue(); // right-
    hand side of production
21.             var length = right.size();
22.             var params = new SemValue[length + 1];
23.
24.             for (var i = 0; i < length; i++) { // parse right-
    hand side symbols one by one
25.                 var term = right.get(i);
26.                 params[i + 1] = isNonTerminal(term)
27.                         ? parseSymbol(term, endS) // for non terminals: recu
    rsively parse it
28.                         : matchToken(term) // for terminals: match token
29.                 ;
30.             }
31.
32.             if(!haveError) act(actionId, params); // do user-
    defined action
33.             return params[0];
34.         }
```

以上就是 PA1B 的工作简介。

2 思考题

Q1.虽然这里 LL1 文法会产生冲突，冲突如下：

$$PS(ElseClause \rightarrow < empty >) \cap PS(ElseClause \rightarrow ELSE\ Stmt) = \{ELSE\}$$

但是在代码中我们认为设定了优先级顺序，即优先产生 ELSE，这样对于每个 IF 都尽量匹配 ELSE，于是问题解决。

Q2.操作符优先级：按数字排序，Op1-Op7 按照操作符的不同优先级归类，同时每个操作符对应的 Expr 也相应划分等级。举例而言，+和-比*和/的优先级低，因此+-都是 Op5 级别而*/是 Op6 级别，相应的 Op5 与 Op6 也只在 ExprT5 和 ExprT6 中出现。

结合性：每次解析都是从左到右解析，但左结合的式子每次解析都解析出符号+表达式。例如 a-b-c，先解析出 a，然后解析出-b（这里，ExprT 会被变为 op Expr ExprT，然后 thunklist 加入了 op 和 Expr，留下 ExprT 继续被解析），最后解析出-c，这样完成了-的左结合解析。

Q3. 考虑下面的代码，唯一的错误是在 int f()后忘了一个大括号。

```
1.  class Main {
2.      int f(){
3.          return 1;
4.      int func(){
5.          int k = 1;
6.          return k;
7.      }
8.  }
```

编译器输出的错误为：

```
1.  *** Error at (4,10): syntax error
2.  *** Error at (4,11): syntax error
3.  *** Error at (5,9): syntax error
4.  *** Error at (6,3): syntax error
5.  *** Error at (8,1): syntax error
```

编译器没有匹配到 f 的大括号，于是认为接下来的部分仍处于 f 函数的定义中，因此错认为 func 函数的所有语句都有语法问题——实际上 func 函数是毫无问题的。编译器可以识别出大括号未匹配的错误，但它在发现大括号未匹配以前，并不知道用户的函数希望在什么时候结束，只能认为它还在函数内部，所以这样的误报实际上是无法避免的。

3 收获

这次试验难度比较大，做完本次实验以后，我对框架的实现以及 LL1 文法的理解有了更深入的理解，这对接下来的实验将有所帮助。