

PA1-B 实验报告

2017011307 张晨

实验原理

LL(1) 语法分析

PA1-A中已经完成了对 `Tree.java` 的修改，此阶段实验仅需要补写文法，但需要保证文法是LL(1)的。

- Abstract

抽象类/抽象函数都以abstract开头，因此加入这些feature后可以很容易地保证LL(1)。需要留心的是，抽象函数需要模仿static，在文法最后加一个fieldList以允许类中同时声明多个函数/变量。

- var

var类型变量与普通类型变量的最大区别是var类型的变量必须初始化。因此在实现中将其视为一种特殊的变量类型，不归入Var里。为尽可能模仿Var的定义，我定义了TypeVAR与VarVAR，分别表示var类型的数据类型和变量；并将Initializer中的 '=' Expr 提出来作为 InitializerNotNull。因为var类型都以"var"开头，这里的LL(1)文法没有太多坑。

- 函数类型

与 type '[' ']' 在文法上很像，所以可以模仿它进行左递归消除。数组类型在框架中的原有定义是 AtomType ArrayType，我将它重命名成了 AtomType AfterAtomType，并在新的 AfterAtomType 中添加 '(' TypeList ')'。在将 ArrayType 改成包括函数类型的 AfterAtomType 后，不再能不能简单的用计数的方式记录数组的维数。我的实现方式是将函数类型、数组类型都放到 thunkList 中，并定义了 buildFuncType 来融合 AtomType 和 AfterAtomType 中的 thunkList。LL(1)的 TypeList 只需要相应地模仿 VarList。

此外，为了区分Expr语法规则 'new' type '[' expr ']' 中的 [是否是最后一维，原有框架在此处也将type写成 AtomType '[' AfterLBrack 和 AfterLBrack : ']' '[' AfterLBrack Or Expr ']'。我用与上一段类似的方法将此部分文法进行改写。

- lambda表达式

实现PA1-A中的lambda表达式的确十分麻烦，但只要按照实验说明中的妥协版文法，这部分可以轻松实现，只需注意需要提取公因式以保证LL(1)

- Call

原有框架中实现调用的方法是 Id ExprListOpt，其中 ExprListOpt 使用非空分支。我首先将所有的 Id ExprListOpt 改成仅含 Id，以去掉框架中原有的所有函数调用。函数调用的原有文法存在左递归，需要用课上所述方法转换。但问题在于，Expr本身是一个十分复杂的东西，生搬理论十分困难。观察测例 call11.decaf，我发现函数调用 '(' ... ')' 和数组 '[' ... ']' 在语法上几乎是等价的，所以我将函数调用放在了和数组一层的位置上，即 ExprT8。

错误恢复

在LLParser的 parseSymbol 中实现实验说明中的算法。

传入函数的 `follow` 表示这个点的祖先的 `follow` 的并集，在此基础上并入当前的 `follow`，就能得到 `end` 集。生成的 `LLTable.java` 中有 `beginSet` 这个函数，调用即可获得 `begin` 集。

错误恢复的方法是，不断向后读 `token`，如果遇到 `begin` 集的，停止往后读，开始进行分析；如果遇到 `end` 集的，停止分析当前 `symbol`，返回 `null`。

为避免抛出空指针异常，只有在当前节点及子节点分析都成功的情况下才执行 `act`。

完成以上几点后，即可找出所有的出错位置。但是，每个出错位置都会多次报错。解决方法是封装 `yyerror` 和 `nextToken` 进行手动去重。

问题回答

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

在 `ElseClause` 中，`ELSE` 分支比 `empty` 分支的优先级高。因此在解析非终结符的过程中，遇到 `else` 会选择 `else` 分支，使得它与最近的 `if` 匹配。

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**。

以熟悉的加减乘除操作为例。

框架将二元运算符分成了 `Op1-Op6`，标号越大优先级越高，在同一个 `Op*` 中的优先级相同。如 `Op5` 为加减，`Op6` 为乘除模。

解析加减乘除的核心文法是

```
1 Expr5      : Expr6 ExprT5
2 ExprT5     : Op5 Expr6 ExprT5
3           | /* empty */
4 Expr6      : .....
```

假设表达式中仅含加减乘除，则 `Expr5` 为加减乘除都可能存在的表达式，`Expr6` 为不含加减的表达式。若不考虑 LL(1) 的要求，可将上述文法写为下述无二义文法。

```
1 Expr5      : Expr5 Op5 Expr6
2           | Expr6
3 Expr6      : .....
```

优先级在 `Expr5` 中的体现是，如果存在一个加减运算符，则将算式分成符号左边、符号右边两部分，分别进行运算，最后再进行 `Op5`。

结合性不体现在文法规则上，而体现在抽象语法树的生成上。框架会将同一优先级的所有子表达式自左至右顺序存到一个 `thunkList` 中，再按照从左至右的顺序取出生成语法树，因而生成的语法树是左结合的。

3. 无论何种错误恢复方法，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。

测例中的 `abstract1.decaf`

```
1 class Main {
2     abstract int v;
3     static void main() { }
4 }
```

我的文法中，对abstract的描述是

```
1 | FieldList => ABSTRACT Type Id '(' VarList ')' ';' FieldList
```

在v匹配完Id后，用分号匹配左括号报错，于是又不断的读static, void, main，直至读到 '(' 才完成匹配，因此报错信息为：

```
1 | *** Error at (2,19): syntax error
2 | *** Error at (3,21): syntax error
3 | *** Error at (3,24): syntax error
4 | *** Error at (4,1): syntax error
```

为了LL1，第2行和第3行的代码被写在了一个非终结符里。故无法通过“遇到end(A)的字符便跳过”的方法进行错误恢复。而该算法中，又没有跳过symbol去match token的做法，故只能一直往后读，直到main后面的 "("。