编译原理PA1-B实验报告

提前上学2018 李嘉图

2019 年 1 月 7 日

1 主要工作

本阶段主要有两个任务：一是为已经实现的基于LL(1)的递归下降分析算法增加错误恢复功能，二是增加新的语言特性。

1.1 错误恢复

错误恢复功能使用一种介于应急恢复和短语层恢复之间的方法。设

$$\begin{aligned} Begin(A) &= \{s|M[A,s] \neq \phi\} \\ End(A) &= \bigcup_{i \in Fa(A)} Follow(i) \end{aligned} \tag{1}$$

其中$Fa(A)$是指分析到当前位置的AST上，A的所有祖先节点。

跳过一些不属于$Begin(A) \cup End(A)$中的字符，如果遇到了$Begin(A)$中的字符，说明程序中可能加入了一些错误的串，从当前位置开始重新分析A；如果遇到了$End(A)$中的字符，说明程序中可能漏掉了一些内容，这时返回`null`表示这里分析出错了，会继续分析之后的内容。

实现这种错误恢复方法，需要在表项为空时不断地读入字符，并检查是否表项不为空，或者在$End(A)$中。实现$End(A)$的方法即是在每次将传入的$follow \cup Follow(A)$作为参数，传入下次分析的调用。

1.2 新增文法的处理

1.2.1 Scopy和Sealed语句的支持

Scopy和Sealed的文法是LL(1)的，且不会产生新的左公因式，将PA1-A中`frontend/Parser.y`的文法和对应的动作复制到`frontend/Parser.spec`中即可。

1.2.2 串行条件卫士语句

支持串行条件卫士语句有两个难点：一是会和`if`产生左公因式，二是可能产生的左递归问题。第一个问题需要将`if`语句提出一个`IF`的左公因式，将文法变成：

```
Stmt :  ...  | IF IfAndGuarded
IfAndGuarded :    IfStmt | GuardedStmt
```

第二步消去GuardedStmt文法的左递归，即：

```
GuardedStmt :  '{' IfBranch '}'
IfBranch :  IfSubStmt GuardedRest
GuardedRest :  GUARDED IfSubStmt GuardedRest
IfSubStmt :  Expr ':'  Stmt
```

语义动作使用一个List将所有的子语句收集起来即可。

1.2.3 简单的自动类型推导

直接加入文法：

```
SimpleStmt :  VAR IDENTIFIER '=' Expr | ...
```

并建立对应的节点即可。

1.2.4 数组若干操作

对于数组常量，工作和PA1-A一样，这里不再赘述。

对于数组初始化和数组连接表达式，其关键在于处理结合性和优先级。使用LL(1)文法处理优先级的方法是，设所有的二元运算符为$a_0, a_1, \ldots a_k$，优先级依次递减，用T_i表示只包含前i个符号的表达式的文法，则有：

T_i : $T_i\ a_i\ T_{i-1} \mid T_{i-1}$

但这样会有左公因式和左递归的问题，考虑做如此修改：

T_i : $T_{i-1}\ TRest_i$

$TRest_i$: a_i T_{i-1} $TRest_i$ | ϵ

对于左结合的运算符，需要用一个List收集所有的操作数和运算符，并最后一并建立AST。

对于取子数组表达式和数组下标动态访问表达式，重要的问题是处理和数组下标运算形成的二义性，这可以通过多次提取左公因式，并利用语义值来判断应当如何建立AST。

```
ExprT8 : '[' Expr IndexOrSeq
IndexOrSeq : ']' IndexRest | ':' Expr ']'
IndexRest : ExprT8 | DEFAULT Expr8
```

DEFAULT("default")后是跟Expr8，是因为default语句是优先级最高的运算符。

数组产生式的文法是LL(1)的，不需要多余处理。

2 悬空else问题

Decaf和许多语言都存在所谓的“悬空else”问题，即：

```
if (A)
    if (B)
        S1;
    else
        S2;
```

这里else语句无法确定应该对应哪一个if语句。悬空else的一种简单解决策略是：每一个else都匹配尽可能靠近的if。这样使用LL(1)分析方法时会非常自然，例如文档指出我们的分析程序在存在冲突时按照定义顺序的优先级分析，那么只要按照：

```
IfStmt : IF '(' Expr ')' stmt ElseStmt
ElseStmt : ELSE Stmt | ϵ
```

在分析到elsestmt并读取到lookahead = ELSE时，产生了冲突：

PS(ElseStmt->ELSE Stmt) $\cup$ PS(ElseStmt->ϵ) = $\{$ELSE$\} \neq \phi$

这时根据约定，会按照第一个产生式进行分析，即可以实现需求。

3 数组产生式文法

数组产生式文法如果采用原来的文法，注意到：

`ArrayConstant : '[' Constant (',' Constant)* ']'`

`ArrayComp : '[' Expr FOR IDENTIFIER In Expr [WHILE Expr] ']'`

其左公因式是`'[' Constant`。如果要修改这一点，需要对`Expr`的结构做大幅度改动，其动作语句也要修改，工作量很大。

4 误报分析

在实验实现的错误处理机制中，每次递归都会将上层的`follow`集合并在一起，这样会导致一个明显的问题：在当前非终结符的`follow`集合中的元素，是其在产生式中的某个出现下一个可能的字符，但不一定是当前分析路径上的可能的下一个字符。举例而言，程序：

```
class Main {
    static void main() {
        int i;
        for (i = 0 if ; ; ) {
        }
    }
}
```

报错信息是：

```
*** Error at (7,13): syntax error
*** Error at (7,16): syntax error
*** Error at (7,20): syntax error
```

在分析到`i=0.if`时，`if`不在当前的预测分析表中，而在`SimpleStmt`的`follow`集合中。但很明显，根据上下文，这里后面一定要跟随一个`';'`。在实现的简单算法中，编译器会认为是出现了末尾漏打分号的错误，开始分析之后的`if`语句，导致后面的连环报错。