

编译原理 PA1-B 实验报告

计71 郑逢时 2016012177

编译原理 PA1-B 实验报告

1. 完成过程
 - 1.1 实现 Abstract 特性
 - 1.2 实现 Var 特性
 - 1.3 实现 Lambda 表达式
 - 1.4 实现函数类型
 - 1.5 实现表达式调用
 - 1.6 错误与恢复
2. 所遇挑战及解决方案
 - 2.1 函数类型解析的动作实现
 - 2.2 NewArray 对函数类型失效的解决
3. PA 问题回答
 - Q1
 - Q2
 - Q3
4. 致谢

1. 完成过程

和PA1-A的LALR parser generator不同，本次PA是基于一种通常来说（当然不严格成立）比LALR型文法稍弱一些的LL(1)文法的，从原理上说自顶向下分析的能力一般较自底向上弱，所以要想达到和LALR框架相同的分析能力，我们需要人为参与文法的设计，以避免LL(1)文法的不足。比如要手动修改文法，消除左递归和左公共因子。又比如，由于框架不再支持结合性和终结符的优先级，需要手动设计合适的文法和非终结符等来“定义”优先级。

同时，LL(1)文法在构建抽象语法树的过程中很多时候需要我们递归地、逆向地实现语法动作过程（action），编程难度比第一次PA高很多，这点会在下文中充分体现。

1.1 实现 Abstract 特性

此特性的实现较为简单，只需注册Abstract终结符，在ClassDef相关部分引入新文法即可：

```
#[rule(ClassDef -> MaybeAbstract Class Id MaybeExtends LBrac FieldList RBrac)]
...
#[rule(MaybeAbstract -> Abstract)]
...
#[rule(MaybeAbstract ->)]
...
```

由于新加入和修改的产生式的预测集都无交，这样的设计不会造成冲突，可行。

对于抽象方法，设计方案也和PA1-A如出一辙，不再赘述。

```
#[rule(FieldDef -> Abstract Type Id LPar VarDefListOrElse RPar Semi)]
...
```

这样实现后，能通过一半的 abstract 公开测例，未通过的一半是因为没有实现错误处理与恢复。

1.2 实现 Var 特性

该部分的实现也和PA1-B大相径庭，先注册Var终结符，再新引入左端是Simple的产生式即可：

```
#[rule(Simple -> Var Id Assign Expr)]
...
```

同样，由于Var是新引入的符号，这个产生式的预测集 {Var} 显然不会与其他Simple的产生式冲突，设计自动合理。

实现后，能通过2/3 的 var 公开测例，未通过的一半同样是因为尚未实现错误处理与恢复。

关于lambda特性的实现较为复杂，这里分lambda表达式、函数类型、和表达式调用三个部分的实现来叙述。

首先注册新的终结符 `=>`，`fun`

1.3 实现 Lambda 表达式

根据新提案的描述，识别lambda表达式的优先级应该低于其他任何Expr，故可在框架的 Expr 附近插入新的文法。（根据rust框架的规则，Expr是最低优先级的表达式，Expr1~Expr9的优先级依次升高）

$$Expr \rightarrow fun \ '(\ VarDefListOrElse \ ')\ (\Rightarrow Expr \mid Block)$$

显然，这两个产生式的PS集合都有 'fun'，故我们需合并左公共因子，引入新的非终结符 LambdaBody 以化解冲突。

修改后的文法是：

```
#[rule(Expr -> Fun LPar VarDefListOrElse RPar LambdaBody)]
...
#[rule(LambdaBody -> Arrow Expr)]
...
#[rule(LambdaBody -> Block)]
...
```

显然，后两个产生式的PS集合无交，第一个产生式的PS集合是 {Fun}，必不会与其他文法冲突。

其中, LambdaBody对应的数据类型是

```
pub enum LambdaKind<'a> {  
    Block(Box<Block<'a>>),  
    Expr(Box<Expr<'a>>),  
}
```

用于表示可为表达式、可为语句块类型的Lambda表达式体。

1.4 实现函数类型

新提案规定, 现在的类型 Type 可以产生函数类型, 新增后的文法为

$$\begin{aligned} \text{Type} &\rightarrow \text{BasicType} \mid \text{Type} \text{ '[' } \text{' } \text{' } \text{' } \mid \text{Type} \text{ '(' } \text{TypeList} \text{ ')' } \\ \text{BasicType} &\rightarrow \text{Int} \mid \text{Bool} \mid \text{Void} \mid \dots \end{aligned}$$

这显然包含左递归, 我们用通用方法修改有关 Type 的产生式, 反复消除左递归:

函数类型: 较难, 先对所有(旧的和新增)产生式作整理:

$$\begin{aligned} T &\rightarrow B \quad (B \rightarrow \text{Int} \mid \text{Bool} \mid \dots) \\ T &\rightarrow T[] \mid T(L) \quad (L: \text{TypeList}) \end{aligned} \quad \left. \vphantom{\begin{aligned} T &\rightarrow B \\ T &\rightarrow T[] \mid T(L) \end{aligned}} \right\} \text{用一般方法消左递归}$$

\Rightarrow

$$\begin{aligned} T &\rightarrow BQ \\ Q &\rightarrow []Q \mid (L)Q \mid \varepsilon \end{aligned} \quad Q: \text{Array Or TypeList.}$$

最后:

$$\begin{aligned} L &\rightarrow TM \mid \varepsilon \\ M &\rightarrow , TM \mid \varepsilon \end{aligned}$$

\uparrow
Int (Void, Bool)

可见, 新得到的文法都是无冲突的, 可行。上图中的Q在程序中起名为ArrayDimOrTypeList, 因为它既可以生成很多的[], 也可以生成很多的'(...)'

程序中新实现的文法如下:

```
// Type -> BasicType Q  
#[rule(Type -> BasicType ArrayDimOrTypeList)]  
...  
// Q -> ( TypeListOrEmpty ) Q  
#[rule(ArrayDimOrTypeList -> LPar TypeListOrEmpty RPar ArrayDimOrTypeList)]  
...
```

```

// Q -> [ ] Q
#[rule(ArrayDimOrTypeList -> LBrk RBrk ArrayDimOrTypeList)]
...
// Q -> eps
#[rule(ArrayDimOrTypeList ->)]
...
// L -> T M
#[rule(TypeListOrEmpty -> Type TypeListRem)]
...
// L -> eps
#[rule(TypeListOrEmpty ->)]
...
// M -> , T M
#[rule(TypeListRem -> Comma Type TypeListRem)]
...
// M -> eps
#[rule(TypeListRem ->)]
...

```

这几个文法对应的语法动作实现比较复杂，在第 2. 节详细探讨。

以上文法都实现且保证无bug后，仍有相关测例不能通过 `lambda_func_array` 这个测例，原因是涉及 `new` 关键字，且表达式以 `[expr]` 结束，这里的 `[` 可能会引起冲突。这个问题也在下一章节探讨。

```
var a1 = new int[(int)(int)[5];
```

1.5 实现表达式调用

在引入函数类型后，现在从语法层面上来说，任意的表达式都是Callable的，比如`1(2)`，`(a + b)(c)`等。

为了支持这样的特性，须引入新的文法 $Expr \rightarrow Expr\ '(\ ExprListOrEmpty)'$ ，同时为了避免冲突，删去原有的 $Expr \rightarrow VarSel\ '(\ ExprListOrEmpty)'$ ，整理得以下文法：

$$\begin{aligned}
 Expr &\rightarrow LValue \mid Expr\ '(\ ExprListOrEmpty)'\mid \dots(others)\dots \\
 LValue &\rightarrow Expr\ '['\ Expr\]'\mid VarSel \\
 VarSel &\rightarrow MaybeOwner\ Id \\
 MaybeOwner &\rightarrow Expr\ '.\'\mid \epsilon
 \end{aligned}$$

这显然包括很多左递归，为了简化，先将 `LValue`, `VarSel`, `MaybeOwner` 等所有非终结符的产生式带入第一行，得到单纯的关于 `Expr` 的产生式：

$$Expr \rightarrow Expr\ '['\ Expr\]'\mid Expr\ .\ Id \mid Expr\ '(\ ExprListOrEmpty)'\mid Id \mid \dots(others)\dots$$

其中，上式右端的 `others` 部分rust框架已基本帮我们实现好了，并引入了从 `Expr1` ~ `Expr9` 等非终结符以消除冲突。消除左递归，并代入框架已有的符号后，上面的文法最终化简为：

```
#[rule(Expr8 -> Expr9 Term8)]
...
#[rule(Term8 -> LBrk Expr RBrk Term8)]
...
#[rule(Term8 -> Dot Id Term8)]
...
#[rule(Term8 -> LPar ExprListOrEmpty RPar Term8)] // new rule
...
#[rule(Term8 ->)]
...
```

其中，大部分文法和分析动作都已被框架实现好了，我们只新加入了第四行所示的文法，它恰好对应于表达式调用的新特性。

为了实现新的语法动作，我们需关注符号 Term8 对应的数据结构 `Vec<IndexOrIdOrCall<'p>>>`。它简单地将自顶向下分析过程中扫描到的所有 '[' , '(' ...) 还有 '.Id' “懒惰地”收集起来，最后在 `Expr8 -> Expr9 Term8` 这条规则处处理，自底向上地组建出一连串由索引、调用和开域操作构成的语法树。

新引入的 `Term8 -> LPar ExprListOrEmpty RPar Term8` 这条文法，显然是一个只含参数而不含调用者的 Call，而 `IndexOrIdOrCall` 这个结构中并没有这种枚举类型，因此我拓展了其数据类型：

```
pub enum IndexOrIdOrCall<'p> {
    Index(Loc, Expr<'p>),
    IdOrCall(Loc, &'p str, Option<(Loc, Vec<Expr<'p>>>)>),
    Call(Loc, Vec<Expr<'p>>>), // new
}
```

这样就能很方便地收集和 `exprCall` 有关的语法了。最后在统一构建的函数 `merge_idx_id_call` 中新增对这种类型的构建方法，可类比附近的代码如法炮制，略。

以上实现后，编译会报冲突警告，大概是 `IdOrCall` 的产生式预测集合都有 '('，这是因为新引入的表达式调用文法使得 `IdOrCall` 的 Follow 集合中有左括号。现在不需要这些文法了，删除以下内容：

```
#[rule(Expr9 -> Id IdOrCall)]
#[rule(IdOrCall ->)]
#[rule(IdOrCall -> LPar ExprListOrEmpty RPar)]
```

并引入：

```
#[rule(Expr9 -> Id)]
```

编译就不再报冲突警告。

实现以上特性，解决所遇到的问题后，能通过除了 syntax error 测例以外的所有测例。

1.6 错误与恢复

有了框架的基本实现，加上PA文档的算法指点，实现错误与恢复是直截了当的。主要的代码及实现思路如下：

```
let (prod, rhs) =
  if let Some(x) = table.get(&(lookahead.ty as u32)) {
    x
  } else {
    // *** handle error and recover ***
    self.error(lookahead, lexer.loc());
    // try if la in End(A)
    if end.contains(&(lookahead.ty as u32)) { return StackItem::_Fail; }
    loop {
      *lookahead = lexer.next();
      if let Some(x) = table.get(&(lookahead.ty as u32)) { // la in Begin(A)
        break x; // recover analysing A
      }
      if end.contains(&(lookahead.ty as u32)) { // la in End(A)
        return StackItem::_Fail; // failed A, continue analysing
      }
    }
  }
};
```

最外层if的 then分支是在读取lookahead（以下简称la）后能从预测分析表中查询到产生式的情况，我们需要实现else分支，也就是没有找到相应的产生式的情况。首先框架会调用报错函数，输出一个 syntax error，接下来进入“核心态”，程序需要向前继续读取符号，直到出现End(A) U Begin(A) 中的符号。

这里的一个小细节是，当前引起错误的这个符号有可能就在 End(A) 中，这时候无需继续向前看，即可结束对当前非终结符A的分析，这样一个符号也不跳过就可以恢复。

否则就持续向前看，如果找到 Begin(A) 中的符号就重新开始对A的分析，如果找到 End(A) 的符号就 return，放弃对A的分析（框架保证 EOF 在End(A) 中，故不会死循环）。

2. 所遇挑战及解决方案

2.1 函数类型解析的动作实现

相关文法：（参考1.4节）

- 1 Type -> BasicType Q
- 2 Q -> (TList) Q
- 3 Q -> [] Q
- 4 Q -> ε

难点一方面在于，Q代表的函数类型句子的一个“后缀”，它可以由多个'(...)'和多个'[]'构成，类型不单一；另一方面在于，中间两个文法Q都递归地出现在了产生式的最右端，按照自顶向下分析的顺序，右端Q的语法树先构造出来，再构造左边Q的语法树。

解决方案当然可以仿照框架其他部分的处理，例如用一个Vec存储各种可能的符号，再最后统一merge构建树。但我采用了一种一边分析文法，一边“现场”构建语法树的策略，这样可以实现对之前定义的数据结构 `SynTyKind::FunType` 的复用。

基本思想是，Q虽然只是一个后缀，但我们发现，只要在任何这样的后缀加上一个Var类型的前缀（Var是非法的，但在这里可以表示“待填入”），就能构成一个完整的 `SynTy`，例如 `var()[], var[]()` (`int()`) 等等。于是赋予Q以 `SynTy` 的类型，这是比较自然的。

然后，

- 一旦使用第二种产生式，我们就可以将右边 Q 的最内层的返回值类型替换为一个新的函数类型 `FunType`，其中返回值类型为Var，参数列表为 `TList`；
- 一旦使用第三种产生式，就找到 Q 的最内层的返回值类型，将其的数组维数加一；
- 使用第三种产生式，直接返回一个维数为0，类型是Var的 `SynTy`。

这里的最内层的返回值类型是说，Q如果是函数类型，它的返回值类型也有可能是一个函数类型，返回值的返回值类型亦有可能是函数类型.....我们循环遍历Q的返回值类型，直到无法遍历（不再是函数类型，而是一般的类型）为止，这时候的遍历到的类型叫“最内层返回值类型”。特别的，当Q不是函数类型时，Q自身就叫“最内层返回值类型”。

最后递归到第一种产生式时，将 `BasicType` 类型的位置和类型填入右端Q的最内层返回值类型处，同时沿着遍历路径将沿途的 `SynTy` 的 `loc` 字段都设为 `BasicType` 对应的 `loc`。

2.2 NewArray 对函数类型失效的解决

原有框架的文法是 `NewClassOrArray -> BasicType LBrk NewArrayRem`，所以在遇到 `new` 一个 `int()[5]` 这种函数类型数组的时候会报错。

我先尝试将 `BasicType` 改为 `Type`，但显然这样一来 Q（定义见2.1）的 Follow集合会有 LBrk，这导致2.1节中的第3、4种产生式因 LBrk 而冲突。

这个问题与我而言过于复杂，冥思苦想之后和一些同学沟通（见4.），了解到了一种可行的文法，能够避免Q的Follow集合出现 LBrk，豁然开朗。

这种文法没有将上面的 `BasicType` 改为 `Type`，而是“将计就计”，在 `BasicType` 和 LBrk 之间引入新的非终结符 `ParamLists`，代表形如 `('(' TypeList ')')`* 这种符号串。（很多个括号包起来的类型列表）

解析过程中，后续的 `NewArrayRem` 中也可能会出现这种 `ParamLists`，所以也要修改其生成文法，最终得到以下LL(1)型文法：

```
NewClassOrArray -> BasicType ParamLists LBrk NewArrayRem | ...others...
NewArrayRem -> RBrk ParamLists LBrk NewArrayRem | Expr RBrk
ParamLists -> LPar TypeListOrEmpty RPar ParamLists | ε
```

对 `ParamLists`、`NewArrayRem` 相关语法动作和数据结构方面的实现，可以仿照2.1来实现，不再赘述。

实现了所有特性，解决了以上困难后，我的decaf编译器最终能通过除了abstract1/3, lambdabad1的所有测例，包括multi-errors ✓

之后解决了一个有关abstract1有关的错误误报问题，大体思路是改进和abstract method相关的文法，将 `(' VarDefList ')` 整体打个包，引入新的非终结符 `ParenedVarDefList` 替代之，实现“整体认读”，这样当遇到 `abstract int a;` 这种符号串的时候，读取到 `;` 时会直接让 `ParenedVarDefList` 出错并恢复（因为`;`在其End集合中），避免了误报。

3. PA 问题回答

Q1

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

空悬else是LL(1)文法无法解决的一个障碍，因为 `MaybeElse -> Else Blocked` 和 `MaybeElse ->` 这两个产生式的预测集合都有 Else。通常程序语言的解决方案都是规定在遇到Else的时候，采取第一种产生式，否则采用第二种，相当于最近邻匹配。本框架也不例外，它将第一种产生式定义在前面，这样默认就会优先采用第一条。

Q2

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。

可以将从Expr衍生出多个非终结符，比如本框架中的 `Expr1 ~ Expr7`，分别对应 `or(Op1)`, `and(Op2)`, `eq/ne`, `lt/le/ge/gt`, `add/sub`, `mul/div/mod`, `not/neq(Op7)` 这几种运算符。优先级最低的 Expr 可以生成 `Expr1`，`Expr1`又可以生成`Expr2 Op1 Expr2 Op1 Expr2...`，`Expr2` 可以生成 `Expr3 Op2 Expr3 Op2 ...` 以此类推。

所以自顶向下的分析过程，会让Expr试图从Expr1变到Expr7，递归地向下分析，而Expr下标数字越大的产生式会越先完成分析，递归上溯。这样就做到了下标数字越大，优先级越高。

例如，`"a + b * c"` 这个符号串的分析过程是：

```
Expr -> Expr1
Expr1 -> Expr2 Term1
...
Expr5 -> Expr6 Term5
...
Expr9 -> Id (match a)
Term8 ->
...(reduce to Expr6 == a)
Term5 -> Op5 Expr6 Term5
Op5 -> Add (match +)
Expr6 -> Expr7 Term6
...
```



```

Expr9 -> Id (match b)
...(reduce to Expr7 == b)
Term6 -> Op6 Expr7 Term6
Op6 -> Mul (match *)
...
Expr9 -> Id (match c)
...(reduce to Expr7 == c)
Term6 ->
(reduce to Expr6 == b * c)
Term5 ->
(reduce to Term5 == + b * c)
...(reduce to Expr5 == a + b * c)
...(finally reduce to Expr == a + b * c)

```

可见，`b*c` 部分先规约，`a+(b*c)` 后规约。

对于结合性，由于多数运算符都是左结合的，而以上文法设计会导致分析的递归上溯的过程是“从右到左”扫描的，相当于右结合。比如对“1+2+3”，会得到 {+3, +2, 1} 这样的符号栈。为了实现左结合，框架专门设计了 `merge_terms` 方法，现将栈倒序过来，递归地构建运算树，得到 ((1+2)+3) 这样的正确结果。

Q3

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。

例如，`abstract3.decaf` 这个测例：

```

class Main {
    abstract void foo() { }
    static void main() { }
}

```

我的程序运行结果是

```

*** Error at (2,25): syntax error
*** Error at (3,5): syntax error

```

即，第二行的 `{}` 处报错，第三行 `static` 处报错。很显然第二处错误误报了。

然而这种误报很难避免，因为在我们的错误恢复算法流程中，一切都正常执行，直到扫描到第一行的 `{` 的时候，此时分析到的非终结符是 `FieldDef`，正在使用 `FieldDef -> Abstract Type Id ParenedVarDefList Semi` 这个产生式，而且分析到的右端项是 `Semi`，也就是分号这个终结符。按照算法，

1. `Semi` 与 `{` 失配，应该抛出一个错误，这没问题，但接下来算法采取的行为是

2. 跳过了当前正在分析的Semi，保留 {，放弃了对FieldDef的分析；
3. 尝试在上一级，也就是FieldList 的预测分析表中查找 {，没找到，继续进入错误态；（这里不会重复在同一个位置报错）
4. 继续向前扫描，扫描到 }，这刚好在 End(FieldList) 中，于是按照算法，跳过对FieldList的分析，进入对上层ClassDef的分析，匹配掉这个 }；
5. 继续对ClassList分析，扫描到下一行的第一个符号是static，发现也不在ClassList的预测分析表中，再一次报错（造成误报）；
6. 往后扫描的void main等符号也都不在 Begin(ClassList) U End(ClassList) 中，一直扫描到EOF，直到结束也没能恢复。

在以上例子中，算法误报的原因是在匹配Semi和 { 失配时没有跳过第一行的 { }，而是保留了 {，试图对 FieldList作同样的匹配，从而造成一连串的误报。无法避免这种误报是因为 Semi 是一个终结符，终结符在失配时，算法必须保留此时的lookahead符号，不然会造成其他大多数情况的错误恢复出问题。

4. 致谢

实验中向顾掀宇、陈海天同学请教过一些有关LL(1)文法的重新设计和错误恢复的问题。没有他们的帮助我很难完成本次PA，在此对各位表达感谢。🙏