

# 编译原理 PA1-A 实验报告

---

计71 郑逢时 2016012177

## 编译原理 PA1-A 实验报告

### 1. 完成思路

#### 1.0 配置启动部分

##### 1.1 实现 `abstract` 特性

###### 1.1.1 对抽象类

###### 1.1.2 对抽象方法

##### 1.2 实现 `var` 特性

##### 1.3 实现 First-class-function 特性

###### 1.3.1 函数类型的解析

###### 1.3.2 Lambda表达式的解析

### 2. 所遇挑战及解决方案

#### 2.1 `ClassDef` 的输出难以用宏实现

#### 2.2 函数类型造成循环定义

#### 2.3 编译出现与左括号文法相关的warning

### 3. PA 问题回答

### 4. 致谢

## 1. 完成思路

---

### 1.0 配置启动部分

参照李晨昊助教的建议，我clone了整个decaf PA的工程框架，以便今后的PA更加顺利地进行。

浏览了看工程的源文件分布，我了解到：

- `syntax`文件夹下的 `parser.rs` 是本次实验实现的核心部分，包括文法的设计，构建ast的过程都在这里定义和实现；
- `syntax`文件夹下的 `ast.rs` 是和构建抽象语法树相关的数据结构定义，新增规则可能会新增数据结构或枚举，定义和实现可以参考这里原有的代码；
- `print`文件夹加下的 `ast.rs` 是对抽象语法树debug输出的实现，新增的数据结构相应地也需要在这里实现输出方法，否则没法通过测例的检查。

### 1.1 实现 `abstract` 特性

此特性的语法解析较容易实现。

首先在 `parser.rs` 中新增一种终结符： `'abstract' = 'Abstract'`

#### 1.1.1 对抽象类

先在 `parser.rs` 中找到

```
1 | #[rule(ClassDef -> Class Id MaybeExtends LBrC FieldList RBrC)]
```

这一句，将其改为

```
1 | #[rule(ClassDef -> MaybeAbstract Class Id MaybeExtends LBrC FieldList  
  | RBrC)]
```

传入参数中添加是否为抽象类的bool变量，同时需增加关于MaybeAbstract的文法。

修改文法之后，`ClassDef` 的解析规则也要相应地改动，即，在 `ast.rs` 的 `struct ClassDef` 中新增 `abstr_: bool` 这一个域，同时在解析规则中要把这一域和 MaybeAbstract 对应的参数关联上。

```
1 | #[rule(ClassDef -> MaybeAbstract Class Id MaybeExtends LBrC FieldList  
  | RBrC)]  
2 | fn class_def(&self, abstr_: bool, c: Token, name: Token, parent:  
  | Option<'p str>, _l: Token, field: Vec<FieldDef<'p>>, _r: Token) -> &'p  
  | ClassDef<'p> {  
3 |     self.alloc.class.alloc(ClassDef {  
4 |         abstr_, // new field  
5 |         loc: c.loc(),  
6 |         name: name.str(),  
7 |         parent,  
8 |         field,  
9 |         parent_ref: dft(),  
10 |        scope: dft(),  
11 |     })  
12 | }
```

最后，还需更新debug输出部分。这部分的改动在 `print/src/ast.rs` 中，但由于涉及宏的使用，比较困难，放在第2节探讨。

### 1.1.2 对抽象方法

抽象方法比抽象类实现稍困难，主要是抽象方法的body部分不再是block，而是一个分号（或者说没有body），不仅FuncDef的产生式要新增：

```
1 | #[rule(FuncDef -> Abstract Type Id LPar VarDefListOrEmpty RPar Semi)]
```

而且现在开始要允许 `FuncDef` 的 `body` 字段为空：

```
1 | pub body: Option<Block<'a>>
```

但是如此改动，会导致之前所有和 `FuncDef` 相关的代码不兼容，编译会失败。为了通过PA1，我将所有有关 `body` 域的错误代码改成 `.body.as_ref().unwrap()` 这种形式，才能通过编译。

在输出方面，为判断一个方法是否抽象，可以body域是否为依据，在 `FieldDef` 的输出规则中更新：

```
1 | if f.body.is_none() { "ABSTRACT".print(p); }
```

## 1.2 实现 `var` 特性

实现本地类型推导主要应该是后续PA的工作，在PA1-A我们只需实现对关键字var的识别和对相关文法的说明。

首先新增一种终结符： `'var' = 'Var'`

然后新增一种 Simple 的产生式，用于产生本地自动类型推导语句：

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

注意这里的产生式右边必须有赋初值的语句。

同时，新增一种 `SynTyKind` 的枚举类型 `Var`，表示自动推导类型：

```
1 | pub enum SynTyKind<'a> {  
2 |     Int,  
3 |     Bool,  
4 |     String,  
5 |     Void,  
6 |     Var,      // local variable ty deduction  
7 |     Named(&'a str),  
8 | }
```

同时新增对 Var 的 print 实现：

```
1 | SynTyKind::Var => write!(p, "<none>").ignore(),
```

## 1.3 实现 First-class-function 特性

本部分的实现较为复杂，分几个阶段完成。

### 1.3.1 函数类型的解析

First-class-function 首先允许用户在程序中标注函数类型，例如 `int(int, float)`，即需要增加以下产生式：

```
1 | #[rule(Type -> Type LPar TypeListOrEmpty RPar)]      // eg: int(int, float)
```

其中，第一个 Type 表示返回值类型，非终结符 TypeListOrEmpty 表示参数列表对应的类型列表（如  $\epsilon$  | int | int, float | int(int) ...），的相关产生式可以参照 VarDefListOrEmpty 的相关函数，其返回类型被我定义为 `Vec<SynTy<'p>>`。

相应的，我们需要补充 `SynTyKind` 中关于函数类型的定义，我最初的写法是

```
1 FunType((SynTy<'a>, Vec<SynTy<'a>>)),
```

元组的第一项表示返回值类型，第二项是参数类型列表。

但这么写过不了编译，详情在下一节解释。

打印方法的实现：

```
1 SynTyKind::FunType((ret, params)) => {
2     write!(p, "TLambda @ {:?}", self.loc).ignore();
3     p.indent(|p| ret.print(p));
4     p.indent(|p| params.print(p));
5 }
```

### 1.3.2 Lambda表达式的解析

现在允许解析fun开头的lambda表达式，而且lambda的主体可以是单个表达式，也可以是语句块，所以需要设计以下新文法：（）

```
1 #[rule(Expr -> Fun LPar VarDefListOrEmpty RPar Arrow Expr)]
```

```
1 #[rule(Expr -> Fun LPar VarDefListOrEmpty RPar Block)]
```

其中，Arrow 是终结符： `'=>' = 'Arrow'`

显然，我们还需增加一种新的表达式类型。在 `ExprKind` 这一枚举类型中新增：

```
1 Lambda(Lambda<'a>),
```

还需定义新的结构：

```
1 pub struct Lambda<'a> {
2     pub params: Vec<&'a VarDef<'a>>,
3     pub body: LambdaKind<'a>,
4 }
5
6 pub enum LambdaKind<'a> {
7     Block(Box<Block<'a>>),
8     Expr(Box<Expr<'a>>),
9 }
```

其中，`Lambda` 的 `body` 域要么是表达式要么是表达式，要么是语句块。

最后，还要新增表达式Call的文法（因为表达式现在可以是First-class-function）

```
1 | #[rule(Call -> Expr LPar ExprListOrEmpty RPar)]
```

但这一文法会导致编译时出现一个与左括号LPar有关的warning，下一节再分析。

`Lambda` 类的print的方法容易通过宏调用来实现，不再赘述。

实现以上特性，解决所遇到的问题后，能通过所有测例。✅

## 2. 所遇挑战及解决方案

第一次熟悉平台并完成PA，可谓困难重重，举步维艰。这里列举一些比较有典型的、有探讨价值的挑战。

### 2.1 `ClassDef` 的输出难以用宏实现

在 `print/src/ast.rs` 中，框架已定义好若干宏替换，以方便地实现对不同类的打印。例如用 `print_struct!` 这一宏规则定义了 `Program`, `ClassDef`, `VarDef` 等类的输出规则。

但在抽象类的新提案中，要求对抽象类输出 "ABSTRACT" 一词。如果只是在 `print_struct!` (`ClassDef...`) 中新增 `self.abstr_` 域，会得到简单的 true/false，不能得到我们期望的输出。

因此，这里需要特化对新的 `ClassDef` 的输出规则。我的解决方案是注释掉了宏规则，而手动实现 `ClassDef` 的 `Printable` trait。这里可以借助Clion IDE的 `macro expand` 工具快速得到与原来调用宏规则等价的代码，然后稍微添加对 `abstr_` 域的输出规则即可完成实现。

```
1 // Print method for ClassDef is specifically defined
2 impl Printable for ClassDef<'_> {
3     fn print(&self, p: &mut IndentPrinter) {
4         write!(p, "ClassDef @ {:?}", self.loc).ignore();
5         p.indent(|p| {
6             if self.abstr_ { "ABSTRACT".print(p); }
7             (self.name).print(p);
8             (self.parent).print(p);
9             (self.field).print(p);
10        });
11    }
12 }
```

### 2.2 函数类型造成循环定义

最初定义的函数类型过不了编译：

```
1 FunType((SynTy<'a>, Vec<SynTy<'a>>)),
```

原因是 `SynTy` 结构的域中包含了一个 `SynTyKind`，从而造成递归定义。

在和几位同学讨论后，我发现了一种很容易的解决方案：用rust标准库中的 `Box` 模板封装递归部分即可，这么做相当于将 `SynTy` 字段存储在堆上，从而避免了递归定义：

```
1 FunType((Box<SynTy<'a>>, Vec<Box<SynTy<'a>>>)),
```

以后在创建的时候，也用 `Box::new(_)` 方法。

## 2.3 编译出现与左括号文法相关的warning

如题，在定义lambda call的文法后，编译时会出现一个warning，与其他同学仔细分析后发现是LPar左括号的优先级的的问题。

例如：表达式 `fun(int x) => 1(5)`，这里的括号和lambda表达式的优先级就是不确定先后的。

- 如果先移进 '(' 则解析为 `fun(int x) => Call(1, 5)`
- 如果先规约lambda表达式，则解析为 `Call((fun(int x) => 1), 5)`

查阅了一些资料，了解到现代语言一般解析为前一种，这意味着左括号的优先级应该很高，至少高于lambda表达式。

为了解决这个冲突，我们只需定义 LPar 的优先级，将其定义为和 RPar 一样高：

```
1 { assoc = 'no_assoc', terms = ['LPar', 'RPar', 'Empty'] },
```

这样，既不会有编译警告，也能通过上述刁钻的测例了。

## 3. PA 问题回答

1. 有一部分 AST 结点是枚举类型。若结点 `B` 是枚举类型，结点 `A` 是它的一个 variant，那么decaf语法上会不会 `A` 和 `B` 有什么关系？限用 100 字符内一句话说明。

A是B的一个不定型实例，说明在decaf语法上存在产生式，其左端是B，右端是A，或者说符号A是符号B的果实/子节点。

2. 原有框架是如何解决空悬 else (dangling-else) 问题的？限用 100 字符内说明。

采用“合并同类项”的方法消除了原始文法的歧义性，即在If/Else产生式引入中介的非终结符MaybeElse，再让后者产生空或者Else。

3. 输入程序 lex 完得到一个终结符序列，然后构建出具体语法树，最后从具体语法树构建抽象语法树。这个概念模型与框架的实现有什么区别？我们的具体语法树在哪里？限用 120 字符内说明。

框架的实现过程中没有显式地构造CST，而是直接从TokenStream一边解析，一边得到AST，其余流程一致。我们的具体语法树可以体现在：parser的函数递归返回过程中，相当于从下而上地构建出了CST；也可以说，推导parser函数前面宏定义的文法的过程，就从上而下地构建了CST。

## 4. 致谢

---

实验中与顾掀宇、钟闰鑫、王征翊同学讨论过，请教过李晨昊助教有关平台的问题。在此对以上同学和助教表达感谢。🙏