PA1-B Report

计71 钟闰鑫 2017010306

实现内容

抽象类

对于抽象类跟抽象函数，直接添加

```
ClassDef -> Abstract Class Id MaybeExtends LBrc FieldList RBrc
FieldDef -> Abstract Type Id LPar VarDefListOrEmpty RPar Semi
```

仍保持LL(1)文法

局部类型推断

添加`Simple -> Var Id Assign Expr`即可，仍保持LL(1)文法

First-class Functions

函数类型

要在`type`这个非终结符中添加函数类型，在PA-1A中直接添加了`Type -> Type LPar TypeListOrEmpty RPar`以及`TypeListOrEmpty`相关的产生式，这里需要消除左递归

将目标产生式

```
Type -> SimpleType ArrayDim | Type LPar TypeListOrEmpty RPar
TypeListOrEmpty -> TypeList | \eps
TypeList -> TypeList Comma Type | Type
```

消除左递归得到

```
Type -> SimpleType ArrayDim Type0
Type0 -> LPar TypeListOrEmpty RPar ArrayDim Type0 | \eps
TypeListOrEmpty -> TypeList | \eps
TypeList -> Type TypeList0
TypeList0 -> Comma Type TypeList0 | \eps
```

除此之外，原框架中的`new`相关产生式也需要更改以支持函数类型

- 注意到`var x = int ()[]()[]()[10]`，从中可以看出必须读到`10`处(表达式处)时我们才能确定`type`类型已经读取完，由此需要针对`[`和`(`进行不同情况的判断，从而保持LL(1)

将原来的产生式

```
NewClassOrArray -> SimpleType LBrk NewArrayRem
NewArrayRem -> RBrk LBrk NewArrayRem | Expr RBrk
```

修改为

```
NewClassOrArray -> SimpleType NewSuffix
NewSuffix -> LBrk NewSuffix0 | LPar TypeListOrEmpty RPar NewSuffix
NewSuffix0 -> RBrk NewSuffix | Expr RBrk
```

- 其中`NewSuffix`判断`[`和`(`，显然它们PS集合不会冲突，而`NewSuffix0`判断是继续作为类型解析还是解析到`Expr`作为new语句一部分

函数表达式

最终目的是要完成`Expr -> Expr ( ExprListOrEmpty )`产生式

在原框架中对于`x.a`和`x.a()`的处理采用了一个`IdOrCall`的非终结符来表示一个`id`是否后面跟着一对括号(来表示函数调用)，而在对于`x.a().b()[].c.d()`此类的处理中，原框架将其划分为`x.a() .b() [] .c .d()`，构成一个`vec`，然后在`merge_idx_id_call`中从左到右遍历解析出`expr`

现在任何表达式都能call

考虑到函数表达式优先级(或者说`(`的优先级)，在`Term8`中进行更改，添加

```
Term8 -> LPar ExprListOrEmpty RPar Term8
```

- 在`Term8`中添加特性，这样使得诸如`IntLit True False`等(`Expr9`)优先级高于函数调用

`Term8`的数据结构为`Vec<IndexOrIdOrCall>`，里面目前只能有`IdOrCall`和`Index`两种类型，注意到新的函数调用，如`(a + b)()`，和原来的函数调用`function_name()`都会产生一个`Call`的数据类型，而其区别只在于，前者`Call`的`func`属性为表达式`a + b`，后者为`VarSel`的`functino_name`，而`IdOrCall`这个`enum`统一考虑了单纯的`Id`和以`Id`作为函数名的函数调用

对于新特性，由于任何表达式都能call，因此直接考虑把`IdOrCall`分开成`Id`和`Call`，此时`IndexOrIdOrCall`数据结构变成

```
pub enum IndexOrIdOrCall {
    Index(Loc, Expr),
    Id(Loc, &str),
    Call(Loc, Vec<Expr>),
}
```

这样的数据结构更加清晰，举个例子

对于`x.a().b()[].c.d()`，现在被划分为了`x.a () b () [] .c .c ()`，其中分别对应是`Id Call Id Call Index Id Id Call`

然后在`merge_idx_idx_call`里面修改对`Id Call Index`的处理逻辑即可

此时会有一些冲突，主要是原来的`IdOrCall`有冲突，由于现在`Id`，`Call`都被分开，因此`IdOrCall`所有产生式都可以删除，并把原来的`Expr9 -> Id IdOrCall`改成`Expr9 -> Id`(即现在这个产生式只能推导出`Id`)即可

错误恢复

根据指导书提供的算法，在原框架对应处添加如下即可

```
self.error(lookahead, lexer.loc());
//recover
if end.contains(&(lookahead.ty as u32)) {
    return StackItem::_Fail;
```

```
}
loop {
    *lookahead = lexer.next();
    if end.contains(&(lookahead.ty as u32)) {
        return StackItem::_Fail;
    }
    if let Some(x) = table.get(&(lookahead.ty as u32)) {
    break x;
    }
}
```

- 需要注意的一点是，在错误发生时的字符需要被判断是否在`end`集合中(代码3-5行)，若是，则应该直接返回，而不消耗任何的字符

问题回答

Q1

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

本阶段框架与PA1-A类似，相关文法为

```
Stmt -> If LPar Expr RPar Stmt MaybeElse
MaybeElse -> Else Blocked | \eps
```

这里会`MaybeElse`会报一个冲突，框架实现是选择`MayBeElse -> Else Blocked`这个产生式，因此在看到下一个是`else`的时候先使用这个产生式表明将`if`和最靠近的`else`进行组合，这就解决了空悬`else`问题(虽然不满足LL(1)文法)

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。

以加法和乘法为例，在框架中，加法和乘法由不同两个op推出

```
Op5 -> Add
Op6 -> Mul
```

数字越大代表优先级越高，与其相关的表达式则是

```
Expr5 -> Expr6 Term5
Term5 -> Op5 Expr6 Term5 | \eps
Expr6 -> Expr7 Term6
Term6 -> Op6 Expr7 Term6 | \eps
```

可以看到

- 优先级方面，通过层次地导出来处理优先级，先导出优先级低的op(如这里的`Op5`在推导中会比`Op6`先推导出来)，用`ExprN`(`N`数字更大)来表示一个不可以被低优先级操作符获得其内运算分量的整体，这样就使得高优先级的操作符的运算分量一定不会被低优先级操作符提前获取
- 结合性方面，从框架LL(1)文法上看是类似于右递归的方式进行的，对于`a+b+c`串，其推导应该是

```
Expr5 -> Expr6 Term5 -> a Term5 -> a Op5 Expr6 Term 5 ->a + Expr6 Term5 -> a + b Term 5 -> a + b Op5 Expr6 Term5 -> a + b + Expr6 Term5 -> a + b + c Term5 -> a + b + c
```

因此可以看到实际上框架文法是右结合的，运算分量会先被右边的运算符得到，这是考虑到LL(1)文法作出的妥协，而为了使得最终得到的是左结合，框架则用一个`vec`来收集所有的同优先级的运算分量，在推导完毕后统一左到右处理，由此实现了左结合性

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。

考虑下列的`decaf`代码

```
class Main {
    int some_var = {}
    int fun() {
        return 0;
    }
}
```

报错信息为

```
*** Error at (2,18): syntax error
*** Error at (3,5): syntax error
```

第一个错误(2, 18)为给类成员赋值，这是正确的错误汇报

但第二个错误(3, 5)则是误报

主要流程如下

1. 发生错误时，当前解析的非终结符是`FuncOrVar`，`lookahead`为`=`，此时`=`不在`FuncOrVar`的PS集合里，因此报错，考虑之后的输入字符，直到`}`的时候，由于`}`在`End`集合里，此时放弃解析`FuncOrVar`，保留`}`，返回上一级，即`FieldDef`
2. 由于`FuncOrVar`为`FieldDef`产生式最后一个，且失败了，因此再次放弃解析`FieldDef`，返回上一级，即`FieldList`
3. 由于`}`在`FieldList`的`End`集合中，因此继续返回上一级`ClassDef`，并匹配消耗这个`}`
4. 此时`lookahead`为`int`，尝试解析`ClassList`(因为产生式`ClassList -> ClassDef ClassList`，上一个`ClassDef`已解析完)，而`int`不再`ClassList`的PS里，因此又报错(误报)，之后继续扫描

由此可以看出，误报的原因在于，当错误发生时，直到出现`}`，但没有消耗`}`，而直接跳转到上层，因此造成了之后的误报

之所以不能避免，是因为`}`是终结符，不能在这种情况被消耗，否则会导致更多错误