编译原理PA1-B实验报告

实验内容

基于 LL(1) 的语法分析与错误恢复。

1. 修改框架文件，将 PA1-A 中所述的所有新特性对应的文法改写为 LL(1)。
2. 使用一种介于二者之间的错误恢复方法，完成错误恢复。

实验过程

LL（1）文法

abstract关键字

按照PA-1的代码，直接引入abstract关键字仍使得文法为LL(1)。不需要特殊的处理。

var关键字

在`SimpleStmt`的产生式添加`var`关键字。`var`的引入不需要特殊的文法上的处理。

First-class Functions

此处是本次实验的核心，在文法方面的处理相当多。

首先完成函数调用。

添加`ExprT8->'(' ExprList ')' ExprT8`，完成对函数调用的解析，将解析到的参数列表放在`thunkList`中。

随后，在`Expr8`和`AfterLParen`处，从`thunkList`中，完成函数调用的解析。

随后完成`Lambda`表达式。

这部分搬运PA1-A代码即可。需要注意的是，一共有两种形式的`Lambda`表达式，他们具有相同的前缀，因此需要提取公因子，完成解析。

最后完成函数类型。

此部分最为麻烦，我如下设计产生式：

```
Type            ->   AtomType ArrayType TLambdas
TLambdas -> '(' TypeList ')' ArrayType TLambdas|/*empty*/
```

如此设计在于`Lambda`表达式的小括号和数组的中括号可以交替出现。为了解决这个问题，在分析过程中，将其分解为一组一组可选的中括号和小括号，分别作为数组和`Lambda`表达式的部分作为分析，并将临时结果放在`thunkList`中。待`Type`产生式右侧分析结束后，从`thunkList`中构造类型。

随即，`new`表达式直观上需要修改为：

```
AfterNewExpr -> Type '[' expr ']'
```

如此修改后，又产生另一新问题。考虑如下的代码段：

```
var a  = new int[][][2];
```

在推断`new`表达式后面的变量的类型时，首先会推断`Type`的类型。在遇到`[2]`时，会产生文法上的错误，即不确定是进行`AfterNewExpr -> Type '[' expr ']'`的推断，还是继续`Type`的解析。为了解决以上问题，做如下修改（此处解决方法询问了其他同学）：

```
AfterNewExpr->AtomType  NewArrayOrLambdaRem
NewArrayOrLambdaRem->'(' TypeList ')'  NewArrayOrLambdaRem
                        | '[' NewArrayRem
NewArrayRem->Expr ExprList1|/* empty */
```

如上解决了`new`语句的分析问题。

错误恢复

当当前无可用的产生式时，即发生了错误。此时通过不停跳过输入的`token`，直到遇到可用的`token`时，继续恢复。第一种可用的`token`是可与当前`symbol`构成产生式的`token`，此时如原来代码那样，继续分析。当遇到`end`集合中的`token`时，返回`null`，放弃分析。

此外，在分析过程的递归调用过程中，维护`end`集合。

思考题

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

空悬else不是LL1文法可以解决的。本阶段的框架通过产生式优先级的方式解决这一问题。`ElseClause -> ELSE Stmt`的优先级高于`ElseClause -> /*empty*/`，因此else会与更近的if结合。

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。

优先级通过Expr(1|2|3|4...)等的层次结构实现。从低优先级到高优先级，分析书树的层次上由高到低。以`a||b&&c&&d||e`为例。`&&`的优先级高于`||`，因此在`||`这一层次，形成`Expr2||Expr2||Expr2`的语句。随后，`Expr2`各自向下推导（即更高优先级），完成整个分析。

左结合性通过`thunkList`实现。同一层次的表达式经过收集后，例如`Expr2||Expr2||Expr2`收集到的三个`Expr2`，从左到右遍历完成左结合。

右结合比较简单，递归即可。

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：

```
class A{
    void f(){
        int a = ();
    }
}
```

程序运行产生如下结果：

```
*** Error at (3,12): syntax error
*** Error at (5,1): syntax error
```

从运行结果可以看出，();产生的错误扩散到了对整个程序的分析。

无法避免在于这种算法没有考虑如')'、';'、'}'等具有"结束一段语句"的作用的关键字，并利用这些关键字来辅助进行错误分析，防止错误的扩散。