PA1-B 实验报告

工作内容

实验部分

PA1-B 实验要求使用 `lalr1` 添加三个新语法的支持：抽象类、局部类型推断和 First-class functions，并且添加错误恢复的支持，需要做这几个方面的修改：

1. 修改 parser 中的产生式，使得它可以支持新的语法
2. 完善 `_parse` 函数，按照要求实现错误恢复。

抽象类

需要添加新的产生式：

```
1  ClassDef -> MaybeAbstract Class Id MaybeExtends LBrc FieldList RBrc
2  MaybeAbstract -> Abstract
3  MaybeAbstract ->
4  FieldDef -> Abstract Type Id LPar VarDefListOrEmpty RPar Semi
```

其余与 PA1-A 一样。

局部类型推断

需要添加新的产生式：

```
1   Simple -> Var Id Assign Expr
```

其余与 PA-A 一样。

First-class functions

匿名函数

添加产生式：

```
1  Expr -> Fun LPar VarDefListOrEmpty RPar ExprOrBlock
2  ExprOrBlock -> Rocket Expr
3  ExprOrBlock -> Block
```

相比 PA1-A 额外进行了左公因子的消除。由于 PA1-B 进行了额外的约束，所以这里就显得非常简单。

函数类型

添加产生式：

```
Type -> SimpleType ArrayDimOrFunction
ArrayDimOrFunction -> LBrk RBrk ArrayDimOrFunction
ArrayDimOrFunction -> LPar TypeList RPar ArrayDimOrFunction
ArrayDimOrFunction ->
```

用类似于数组类型的方法实现了函数类型的解析。

调用语法

添加产生式：

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

考虑到调用的优先级与字段选择、数组索引是同一级的，所以放在一起。

错误处理

需要更改 `_parse` 函数：

```
  // parse impl with some error recovering, called be the generated `parse` function
  fn _parse<'l: 'p>(&mut self, target: u32, lookahead: &mut Token<'l>, lexer: &mut Lexer<'l>, f: &HashSet<u32>) -> StackItem<'p> {
    let target = target as usize;
    // these are some global variables which may be invisible to IDE, so fetch them here for convenience
    let follow: &[HashSet<u32>] = &*FOLLOW;
    let table: &[HashMap<u32, (u32, Vec<u32>)>] = &*TABLE;
    let is_nt = |x: u32| x < NT_NUM;

    let mut end = f.clone();
    end.extend(follow[target].iter());
    let table = &table[target];
    let (prod, rhs) = if let Some(x) = table.get(&(lookahead.ty as u32)) { x } else {
      self.error(lookahead, lexer.loc());
      loop {
```

```
15          if let Some(x) = table.get(&(lookahead.ty as u32)) {
16            // lookahead in Begin(A)
17            break x;
18          } else if let Some(_x) = end.get(&(lookahead.ty as u32)) {
19            // lookahead in End(A)
20            return StackItem::_Fail;
21          }
22          *lookahead = lexer.next();
23        }
24      };
25      let value_stk = rhs.iter().map(|&x| {
26        if is_nt(x) {
27          self._parse(x, lookahead, lexer, &end)
28        } else if lookahead.ty as u32 == x {
29          let token = *lookahead;
30          *lookahead = lexer.next();
31          StackItem::_Token(token)
32        } else {
33          self.error(lookahead, lexer.loc());
34          StackItem::_Fail
35        }
36      }).collect::<Vec<_>>();
37      self.act(*prod, value_stk)
38    }
```

代码流程和实验要求基本是直接对应的：匹配失败时，进入循环，对每个符号进行判断，如果在 `Begin(A)` 中，就直接返回继续分析；如果在 `End(A)` 中，那就返回分析失败。两个集合的查找分别对于原来代码中的 `table[lookahead]` 和 `end[lookahead]` 。这样就可以通过 S1-LL 的若干测例了。

问题回答

Q1. 本阶段框架是如何解决空悬 else (dangling-else) 问题的?

强行忽略了 PS 冲突的问题，选择了最近匹配的产生式，因为它在代码中靠前。

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。

优先级与结合性是通过类似课本上的构造方法完成的，不过由于涉及到二元、一元、左结合与右结合的问题，比教案上更复杂一些。大致的构造思路：

1. 优先级高的放后面，优先级低的放前面
2. 二元操作符放在对应的 Term 中即 `ExprN -> ExprN1 TermN` `TermN -> Op ExprN1 TermN`
3. 一元操作符直接对应 `ExprN -> Op ExprN` 产生式
4. 一些特殊情况单独处理

简单的例子：

Or 的优先级最低：

```
Expr1 -> Expr2 Term1
Term1 -> Op1 Expr2 Term1
Term1 ->
Op1 -> Or
```

左结合还是右结合在合并的顺序上会不一样：左结合就从左到右合并；右结合就从右到左合并。

紧接着就是 And:

```
Expr2 -> Expr3 Term2
Term2 -> Op2 Expr3 Term2
Term2 ->
Op2 -> And
```

同样优先级的就放在同一级的 Op 中：

```
Op4 -> Lt
Op4 -> Le
Op4 -> Ge
Op4 -> Gt
```

一元的右结合的就直接对应：

```
Expr7 -> Op7 Expr7
```

括号和类型转换比较特殊：

```
Expr7 -> LPar Expr RPar Term7
Expr7 -> LPar Class Id RPar Expr7
```

虽然括号既可以是调用也可以是类型转换，而两个情况分别在不同的优先级下，通过这种写法一样可以达到目的。

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。

一个例子是：

```
class Main {
    void test() {return;

    int main() {
        Print("123");
        return 0;
    }
}
```

它会报错：

```
*** Error at (4,13): syntax error
*** Error at (4,14): syntax error
*** Error at (8,1): syntax error
```

实际上只少了第二行一个大括号，但是解析到第四行的时候，它并不能区分变量的定义和函数的定义，于是连续在两个括号处报错，然后认为后面接了一个 Block 。

问题在于错误恢复有的时候是会跑过头的。例如在遇到第四行的 int 的时候，它首先尝试解析一个 `var ('=' expr)?` ，而不是回退到类的一级去解析 `methodDef` ，这就导致遇到括号的时候已经来不及回退，所以不能正确解析处后面的结构了。