工作内容

添加LL(1)文法

抽象类/方法，局部类型推断

抽象类与局部类型推断无需修改，在对应位置添加相关语法后即为LL(1)；抽象方法需要仿照一般方法的定义方式略微修改即可。

函数类型

函数类型中，原来的语法为：

```
Type -> AtomType ArrayType
AtomType -> ...
ArrayType -> '[' ']' ArrayType | <empty>
```

在添加函数类型并消除左递归后，得到：

```
Type -> AtomType ArrayType
AtomType -> ...
ArrayType -> '[' ']' ArrayType | '(' TypeList ')' ArrayType | <empty>
TypeList -> Type TypeList1 | <empty>
TypeList1 -> ',' Type TypeList1 | <empty>
```

需要考虑的是，`ArrayType`如何携带需要的信息给Type，不能采用原来的数的方法，改用两个`Deque`，一个记录当前单元是'[]'还是'(TypeList)'，另一个记录如果所有TypeList的信息。在这部分，`Deque`当做栈使用。结合测例略微书写可以得知，最左侧的单元作为树叶，最右侧的单元作为树根，而最右侧的单元在分析时最先进入容器，因此它要最后出，即我们需要栈的结构。

Lambda表达式

由于'=>'的优先级最低，因此其应该作为语法树顶层出现。其需要提取公因子，无左递归，需要将：

```
Expr -> Expr1
Expr1 -> ...
```

改为

```
Expr -> Expr0
Expr0 -> 'fun' '(' ParamList ')' AfterLambda | Expr1
AfterLambda -> '=>' Expr | Block
ParamList -> Type Id ParamList1 | <empty>
ParamList1 -> ',' Type Id ParamList1 | <empty>
```

函数调用

经过仔细观察，需要将：

```
ExprT8 -> '.' Id ExprListOpt ExprT8 | ...
Expr9 -> Id ExprListOpt | ...
ExprListOpt -> ExprList | <empty>
```

改为：

```
ExprT8 -> '(' ExprList ')' ExprT8 | '.' Id ExprT8 | ...
Expr9 -> Id | ...
```

然后在`Expr8`和`AfterLParen`中根据`sv.id == null`来判断其是否调用新的构造函数。

NEW新类型支持

这部分也要改，但标准测例中没有，感谢陈海天提供的一份测例。

将：

```
AfterNewExpr -> Id '(' ')' | AtomType '[' AfterLBrack
AfterLBrack -> ']' '[' AfterLBrack | Expr ']'
```

改为：

```
AfterNewExpr -> Id '(' ')' | AtomType TypeLists '[' AfterLBrack
AfterLBrack -> ']' TypeLists '[' AfterLBrack | Expr ']'
TypeLists -> '(' TypeList ')' TypeLists | <empty>
```

即将语法变成：

```
AfterNewExpr = Id '(' ')' | AtomType TypeLists '[' (']' TypeLists '[')* Expr ']'
TypeLists = ('(' TypeList ')')*
```

这样可以消除左递归并提取公因子。调用方法的思路与函数类型类似，不过这里需要将`TypeLists`里的`Deque`转移进`AfterLBrack`中，需要同时用到`removeFirst`和`removeLast`，这也是采用`Deque`的原因。

错误恢复

在`LLParser.java`中修改`Parser::parseSymbol(int symbol, Set<Integer> follow)`方法，并添加全局变量`Boolean hasErr`，使得递归中子分支出现错误后，其祖先分支可以识别并报错。

实现的方法与实验指导书相同，实现时需要理解生成的`LLTable.java`的含义，调用`beginSet(int)`，`followSet(int)`方法获取`beginA`和`followA`。

需要将递归调用的参数从`follow`改成`endA`使得`parseSymbol`的第二个参数可以正确表达。

回答问题

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

查阅https://github.com/paulzfm/ll1pg/wiki/2.-Resolving-Conflicts可知，对于文法G[S]：

```
S -> if C then S E
E -> else S | <empty>
```

由于`PS(E -> else S) & PS(E -> <empty>) = {else}`，此文法并非LL(1)文法。

此工具在自动解决冲突时，会指定一个默认优先级，对于`S -> r1 | r2 | r3`有冲突时，会认为`S -> r1`的优先级最高，`S -> r3`优先级最低，从而解决二义性问题。

在本框架中，会优先结合`E -> else S`，从而消除了二义性问题。

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。

不妨考虑一个只支持'+'和'*'的语法G[E]：

```
E -> E1
E1 -> E2 ET1
ET1 -> '+' E2 ET1 | <empty>
E2 -> Id ET2
ET2 -> * Id ET2 | <empty>
Id -> 'a' | 'b' | 'c'
```

考虑`a+b*c`，有

```
E -> E1 -> E2 ET1 -> Id ET2 ET1 -> a ET2 ET1 -> a ET1
-> a + E2 ET1 -> a + Id ET2 ET1 -> a + b ET2 ET1 ->
a + b * Id ET2 ET1 -> a + b * c ET2 ET1 -> a + b * c ET1
-> a + b * c
```

`b*c`产生自E2（ET2），作为树更低的位置，`a+...`产生自E1（ET1），作为树更高的位置，从而实现了运算符优先级。

再不妨考虑一个只支持'+'的语法G[E]：

```
E -> Id ET
ET -> '+' Id ET | <empty>
```

考虑`a+b+c`，有

```
E -> Id ET -> a ET -> a + Id ET -> a + b ET
-> a + b + Id ET -> a + b + c ET -> a + b + c
```

仅考虑语法树的话，ll(1)的语法树对同一级的符号来说总是右结合的。即如果'+'是一个右结合运算符的话，ll(1)直接生成的语法树就是我们想要的结果。而我们需要左结合（即真实的'+'）时，在框架中储存一个vector，根节点根据vector的内容从左至右建树，从而实现了左结合。

另外，java框架中使用的是`ArrayList<>::add(0, Object)` + `for (var sv : container) {...}`的方式，复杂度较高，考虑到实现左结合的过程，将其改为`Stack<>::push()` + `while(!container.empty) {var sv = container.pop(); ... }`的方式，可以减少其复杂度。这与我函数类型的实现是相同的。

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。

对于`abstract1.decaf`而言，代码及结果如下：

```
class Main {
    abstract int v;
    static void main() { }
}
```

```
*** Error at (2,19): syntax error
*** Error at (3,21): syntax error
*** Error at (3,24): syntax error
*** Error at (4,1): syntax error
```

事实上，只有第一行报错是需要的，其余的都属于误报。因为本恢复程序在试图恢复抽象方法的定义时，读到';'之后报错，并继续寻找'('，直到找到了第三行的'main()'，从而使整个第三行产生错误。

由于本恢复程序是依靠`BeginA`和`EndA`来判断跳过符号的，一旦错误的部分没有属于$BeginA \cup EndA$的元素的话，则会跳过它本不该跳过的部分，例如本例中的';'。

同样的，Decaf.jacc会在匹配节点后调用`$$=????`生成AST，因此AST也与CST同步生成。

以上两点是概念模型与框架的实现的区别。

具体语法树仅在匹配过程中暂时存在于parser，随即会在生成对应的AST后消除。