PA1-A 实验报告

实验前导：注册关键字和运算符，涉及五个文件——

三次实验涉及到关键字 ABSTRACT（抽象类）、VAR（局部变量）、FUN（Lambda 表达式）、ARROW（=>运算符）

src/main/jack/Decaf.jacc:将关键字添加到 token 中

```
%token VOID         BOOL        INT         STRING      CLASS       VAR
%token NULL         EXTENDS     THIS        WHILE       FOR         FUN         ARROW
%token IF           ELSE        RETURN      BREAK       NEW
%token PRINT        READ_INTEGER            READ_LINE
%token BOOL_LIT     INT_LIT     STRING_LIT
%token IDENTIFIER   AND         OR          STATIC      INSTANCE_OF ABSTRACT
%token LESS_EQUAL   GREATER_EQUAL           EQUAL       NOT_EQUAL
%token '+'  '-'  '*'  '/'  '%'  '='  '>'  '<'  '.'
%token ','  ';'  '!'  '('  ')'  '['  ']'  '{'  '}'

%right ARROW
%left OR
%left AND
%nonassoc EQUAL NOT_EQUAL
%nonassoc LESS_EQUAL GREATER_EQUAL '<' '>'
%left  '+' '-'
%left  '*' '/' '%'
%nonassoc UMINUS '!'
%nonassoc '[' '.'
%nonassoc ')' EMPTY
%nonassoc ELSE
```

src/main/java/frontend/parsing/JaccParser.java:

```
case Tokens.ABSTRACT -> decaf.frontend.parsing.JaccTokens.ABSTRACT;
case Tokens.VAR -> decaf.frontend.parsing.JaccTokens.VAR;
case Tokens.FUN -> decaf.frontend.parsing.JaccTokens.FUN;
case Tokens.ARROW -> decaf.frontend.parsing.JaccTokens.ARROW;
```

src/main/java/frontend/parsing/SemValue. java:

```
case Tokens.FUN -> "keyword : fun";
case Tokens.ABSTRACT -> "keyword : abstract";
case Tokens.VAR -> "keyword : var";
```

```
case Tokens.ARROW -> "operator : =>";
```

src/main/java/frontend/parsing/Tokens. java:（注意需要避过单字符的 ASCII 码）

```
38      int ABSTRACT = 31;
39      int VAR = 32;
40      int FUN = 34;
41      int ARROW = 35;
```

src/main/jflex/Decaf.jflex:

```
70  "abstract"          { return keyword(Tokens.ABSTRACT);     }
71  "var"               { return keyword(Tokens.VAR);          }
72  "fun"               { return keyword(Tokens.FUN);          }
73
74      // operators, with more than one character
75  "=>"                { return operator(Tokens.ARROW);        }
```

任务 1：抽象类

1. 任务要求与指导

根据语法规范中的 classDef 和 methodDef 的改动，可知需要对 Tree.java 中的 classDe 类和 methodDef 类进行改动。按照上述过程完成关键字 ABSTRACT 的注册。核心思想是抽象类和静态类的关键字用法相似，只要在对 static 进行定义的地方进行重复操作即可。

新特性 1：抽象类

加入 `abstract` 关键字，用来修饰类和成员函数。例如，

```
abstract class Abstract {
    abstract void abstractMethod();
}
```

语法规范：将原来的

```
classDef ::= 'class' id  ('extends' id)?  '{' field* '}'
methodDef ::= 'static'? type id '(' paramList ')' block
```

变成

```
classDef ::= 'abstract'? 'class' id  ('extends' id)?  '{' field* '}'
methodDef ::= 'static'? type id '(' paramList ')' block
            | 'abstract' type id '(' paramList ')' ';'
```

2. 修改 Modifiers 类加入 ABSTRACT 关键字，仿照 static，设置静态常数 ABSTRACT 并令其为 2，同时增加 isAbstract 函数并修改构造函数。具体代码如图：

```
    public static class Modifiers {
        public final int code;

        public final Pos pos;

        private List<String> flags;

        // Available modifiers:
        public static final int STATIC = 1;
        public static final int ABSTRACT = 2;

        public Modifiers(int code, Pos pos) {
            this.code = code;
            this.pos = pos;
            flags = new ArrayList<>();
            if (isStatic()) flags.add("STATIC");
            if (isAbstract()) flags.add("ABSTRACT");
        }

        public Modifiers() {
            this(0, Pos.NoPos);
        }

        public boolean isStatic() {
            return (code == 1);
        }

        public boolean isAbstract() {
            return (code == 2);
        }

        @Override
        public String toString() {
            return String.join(" ", flags);
        }
    }
}
```

3. 修改 classDef 函数和 methodDef 函数。

（1）对前者，由表达式可知，只需添加关键字 ABSTRACT 即可。将是否是 ABSTRACT 作为参数传入 classDef 的构造函数中，并由此确定 Modifiers 的内容。

```
4.      /**
5.       * Class definition.
6.       * <pre>
7.       *     'class' id {'extends' parent}? '{' fields '}'
8.       * </pre>
9.       */
10.     public static class ClassDef extends TreeNode {
11.         // Tree elements
12.         public Modifiers modifiers;
13.         public final Id id;
14.         public Optional<Id> parent;
15.         public final List<Field> fields;
```

```
16.         // For convenience
17.         public final String name;
18.
19.         public ClassDef(Id id, Optional<Id> parent, List<Field> fiel
    ds, Pos pos, Boolean isAbstract) {
20.             super(Kind.CLASS_DEF, "ClassDef", pos);
21.             this.id = id;
22.             this.parent = parent;
23.             this.fields = fields;
24.             this.name = id.name;
25.             if(isAbstract)
26.                 this.modifiers = new Modifiers(Modifiers.ABSTRACT, p
    os);
27.             else
28.                 this.modifiers = new Modifiers();
29.         }
30.
31.         public boolean isAbstract() {
32.             return modifiers.isAbstract();
33.         }
34.
35.         public boolean hasParent() {
36.             return parent.isPresent();
37.         }
38.
39.         public List<MethodDef> methods() {
40.             var methods = new ArrayList<MethodDef>();
41.             for (var field : fields) {
42.                 if (field instanceof MethodDef) {
43.                     methods.add((MethodDef) field);
44.                 }
45.             }
46.             return methods;
47.         }
48.
49.         @Override
50.         public Object treeElementAt(int index) {
51.             return switch (index) {
52.                 case 0 -> modifiers;
53.                 case 1 -> id;
54.                 case 2 -> parent;
55.                 case 3 -> fields;
56.                 default -> throw new IndexOutOfBoundsException(index
    );
```

```
            };
        }

        @Override
        public int treeArity() {
            return 4;
        }

        @Override
        public <C> void accept(Visitor<C> v, C ctx) {
            v.visitClassDef(this, ctx);
        }
    }
```

（2）对于后者，本质上仍是模仿 static 的操作，观察到 isStatic 作为参数传入构造函数中，于是也设置布尔变量 isAbstract 来指示。此处将 body 设为 optionnal<block>的目的是方便在输出的时候完成抽象方法的特殊性（没有函数体）。

```
/**
     * Member method definition.
     * <pre>
     *     'static'? returnType id '(' type1 id1 ',' type2 id2 ',' ...
')' '{' body '}'
     * </pre>
     * Decaf has static methods but NO static variables, strange!
     */
    public static class MethodDef extends Field {
        // Tree elements
        public Modifiers modifiers;
        public Id id;
        public TypeLit returnType;
        public List<LocalVarDef> params;
        public Optional<Block> body;
        // For convenience
        public String name;

        public MethodDef(boolean isStatic, boolean isAbstract, Id id, T
ypeLit returnType, List<LocalVarDef> params, Optional<Block> body, Pos
pos) {
            super(Kind.METHOD_DEF, "MethodDef", pos);
            //this.modifiers = isStatic ? new Modifiers(Modifiers.STATI
C, pos) : new Modifiers();
            this.id = id;
            this.returnType = returnType;
            this.params = params;
            this.body = body;
```

```
            this.name = id.name;
            if(isStatic)
                this.modifiers = new Modifiers(Modifiers.STATIC, pos);
            else if(isAbstract)
                this.modifiers = new Modifiers(Modifiers.ABSTRACT, pos)
;
            else
                this.modifiers = new Modifiers();
        }

        public boolean isStatic() {
            return modifiers.isStatic();
        }
        public boolean isAbstract() {
            return modifiers.isAbstract();
        }

        @Override
        public Object treeElementAt(int index) {

                return switch (index) {
                    case 0 -> modifiers;
                    case 1 -> id;
                    case 2 -> returnType;
                    case 3 -> params;
                    case 4 -> body;
                    default -> throw new IndexOutOfBoundsException(inde
x);
                };
        }

        @Override
        public int treeArity() {
            return 5;
        }

        public <C> void accept(Visitor<C> v, C ctx) {
            v.visitMethodDef(this, ctx);
        }
    }
```

（3）上述过程体现在 Decaf.jacc 中为（需要注意的是$k 的 k 需要随着输入格式的变化而变化）：

```
ClassDef        :   CLASS Id ExtendsClause '{' FieldList '}'
                    {
                        $$ = svClass(new ClassDef($2.id, Optional.ofNullable($3.id), $5.fieldList, $1.pos, false));
                    }
                |   ABSTRACT CLASS Id ExtendsClause '{' FieldList '}'
                    {
                        $$ = svClass(new ClassDef($3.id, Optional.ofNullable($4.id), $6.fieldList, $2.pos, true));
                    }
                ;
```

```
MethodDef       :   STATIC Type Id '(' VarList ')' Block
                    {
                        $$ = svField(new MethodDef(true, false, $3.id, $2.type, $5.varList, Optional.ofNullable($7.block), $3.pos));
                    }
                |   ABSTRACT Type Id '(' VarList ')' ';'
                    {
                        $$ = svField(new MethodDef(false, true, $3.id, $2.type, $5.varList, Optional.empty(), $3.pos));
                    }
                |   Type Id '(' VarList ')' Block
                    {
                        $$ = svField(new MethodDef(false, false, $2.id, $1.type, $4.varList, Optional.ofNullable($6.block), $2.pos));
                    }
                ;
```

4.心得与困难:

（1）一开始没注意 ABSTRACT 应用在 classDef 和 methodDef 中的微小不同，在 methodDef 中抽象函数没有函数体应以';'结尾，仍然下意识地写成了 block，造成了错误。

（2）开始理解 optional 类在输出格式中的应用。

任务 2：局部类型推断

1. 任务要求与指导:

根据语法规范，可以知道需要对 simplestmt 进行改动。仍然需要先行注册关键字 VAR。根据输出要求可知此处需要用到 Optional 类来使得第一个输出为<none>。

新特性 2：局部类型推断

加入 `var` 关键字，用来修饰**局部变量**。例如

```
class Main {
    static void main() {
        // int i = 0;
        var i = 0; // identical as above
    }
}
```

语法规范:

```
simpleStmt ::= ...
             | 'var' id '=' expr
```

2. 修改 LocalVarDef 类的两个构造方法，使第一个变量变为 Optional<TypeLit>。

```
    public static class LocalVarDef extends Stmt {
        // Tree elements
        public Optional<TypeLit> typeLit;
        public Id id;
```

```
7.          public Pos assignPos;
8.          public Optional<Expr> initVal;
9.          // For convenience
10.         public String name;
11.
12.         public LocalVarDef(Optional<TypeLit> typeLit, Id id, Pos ass
    ignPos, Optional<Expr> initVal, Pos pos) {
13.             // pos = id.pos, assignPos = position of the '='
14.             // TODO: looks not very consistent, maybe we shall alway
    s report error simply at `pos`, not `assignPos`?
15.             super(Kind.LOCAL_VAR_DEF, "LocalVarDef", pos);
16.             this.typeLit = typeLit;
17.             this.id = id;
18.             this.assignPos = assignPos;
19.             this.initVal = initVal;
20.             this.name = id.name;
21.         }
22.
23.         public LocalVarDef(Optional<TypeLit> typeLit, Id id, Pos pos
    ) {
24.             this(typeLit, id, Pos.NoPos, Optional.empty(), pos);
25.         }
26.
27.         @Override
28.         public Object treeElementAt(int index) {
29.             return switch (index) {
30.                 case 0 -> typeLit;
31.                 case 1 -> id;
32.                 case 2 -> initVal;
33.                 default -> throw new IndexOutOfBoundsException(index
    );
34.             };
35.         }
36.
37.         @Override
38.         public int treeArity() {
39.             return 3;
40.         }
41.
42.         @Override
43.         public <C> void accept(Visitor<C> v, C ctx) {
44.             v.visitLocalVarDef(this, ctx);
45.         }
46.     }
```

3.修改 Decaf.jacc 中的表达式。

```
SimpleStmt      :   Var Initializer
                    {
                        $$ = svStmt(new LocalVarDef(Optional.ofNullable($1.type), $1.id, $2.pos, Optional.ofNullable($2.expr), $1.pos));
                    }
                |   VAR Id '=' Expr
                    {
                        $$ = svStmt(new LocalVarDef(Optional.empty(), $2.id, $3.pos, Optional.ofNullable($4.expr), $2.pos));
                    }
```

4.困难与心得:

（1）对于所有有 LocalVarDef 的表达式，都需要将第一个参数改为 Optional.pfNullable 或者 Optional.empty，否则会出现编译不通过的问题。

任务 3：First-class Functions

1°函数类型

1. 任务要求和指导:

需要实现两种新的表达式，分别是类似于构造函数的 type = type(type…) 和 迭代形式的 typeList = (type, type, …)。由于含有正则表达式，参照 varList 等类的写法，添加了 typeList 类以及对应的支持。

函数类型

语法规范:

```
type ::= ...
       | type '(' typeList ')'

typeList ::= (type (',' type)*)?
```

括号左边的是返回值的类型，括号内的是诸参数的类型。

2. 实验步骤

在 SemValue.Kind 中注册:

```
class SemValue {
    enum Kind {
        TOKEN, CLASS, CLASS_LIST, FIELD, FIELD_LIST, VAR, VAR_LIST, TYPE, TYPE_LIST, STMT, STMT_LIST, BLOCK, EXPR, EXPR_LIST,
        LVALUE, ID, TEMPORARY
    }
```

在 AbstractParser.java 中加入定义 svTypes:

```
    protected SemValue svTypes(Tree.TypeLit... types) {
        var v = new SemValue(SemValue.Kind.TYPE_LIST, types.length == 0 ? Pos.NoPos : types[0].pos);
        v.typeList = new ArrayList<>();
        v.typeList.addAll(Arrays.asList(types));
        return v;
    }
```

接着修改 Decaf.jacc。根据表达式的格式，可以用上下文无关文法表达如下:

S -> (S) | A

A -> type | type ‘,’ A

用表达式进行表达如图:

```
TypeList        :   TypeList1
                    {
                        $$ = $1;
                    }
                |
                    {
                        $$ = svTypes();
                    }

TypeList1       :   TypeList1 ',' Type
                    {
                        $$ = $1;
                        $$.typeList.add($3.type);
                    }
                |   Type
                    {
                        $$ = svTypes($1.type);
                    }
```

至此完成了 typeList = (type, type, ···)的表达式。

type = type (type···)的完成需要用到 Lambda 表达式，需要新建一个 TLambda 类。所有新建的类需要现在 Tree.java 中进行注册。同样是在 Tree.Kind 中进行注册。

```
public abstract class Tree {
    public enum Kind {
        TOP_LEVEL, CLASS_DEF, VAR_DEF, METHOD_DEF,
        T_INT, T_BOOL, T_STRING, T_VOID, T_CLASS, T_ARRAY, T_LAMBDA,
        LOCAL_VAR_DEF, BLOCK, ASSIGN, EXPR_EVAL, SKIP, IF, WHILE, FOR, BREAK, RETURN, PRINT,
        INT_LIT, BOOL_LIT, STRING_LIT, NULL_LIT, VAR_SEL, INDEX_SEL, CALL, LAMBDA,
        THIS, UNARY_EXPR, BINARY_EXPR, READ_INT, READ_LINE, NEW_CLASS, NEW_ARRAY, CLASS_TEST, CLASS_CAST
    }
```

然后需要在 Visitor.java 中添加支持：

```
    default void visitTLambda(Tree.TLambda that, C ctx) {
        visitOthers(that, ctx);
    }
```

接着添加类的定义：

```
public static class TLambda extends TypeLit {
    TypeLit returnType;
    List<TypeLit> paramTypes;

    public TLambda(Pos pos, TypeLit returnType, List<TypeLit> paramTypes){
        super(Kind.T_LAMBDA, "TLambda",pos);
        this.returnType = returnType;
        this.paramTypes = paramTypes;
    }

    @Override
    public Object treeElementAt(int index) {
        return switch(index){
            case 0 -> returnType;
            case 1 -> paramTypes;
            default -> throw new IndexOutOfBoundsException(index);
        };
    }

    @Override
    public int treeArity() {
        return 2;
    }

    @Override
    public <C> void accept(Visitor<C> v, C ctx) {
        v.visitTLambda(this, ctx);
    }
}
```

最后添加表达式:

```
| Type '(' TypeList ')'
    {
        $$ = svType(new TLambda($1.pos, $1.type, $3.typeList));
    }
;
```

至此完成了第一个小特性。

3. 困难与心得:

（1）与同学沟通后才意识到新类的建立需要进行注册和支持操作。

2°Lambda 表达式

1. 任务要求与指导:

需要实现 block lambda 和 expression lambda 两种类型的 Lambda 表达式, 需要用到'=>'操作符。提示中已经指出该操作符应该优先度最低，因而在添加关键字的时候需要在前方加上%right 并且置于所有操作符之前。

Lambda 表达式

有两种, block lambda 和 expression lambda。Lambda 表达式的类型是函数类型。

语法规范:

```
expr ::= ...
       | 'fun' '(' paramList ')' '=>' expr
       | 'fun' '(' paramList ')' block

paramList ::= (type id (',' type id)*)?
```

其中新增的箭头操作符 `'=>'` 左边是参数列表，右边是返回值。 `'=>'` 的优先级最低。 `fun` 为新增的关键字。 Block lambda 可以包含 return 语句表示返回值（当然，没有 return 语句的话返回类型是 void）。

2. 实验步骤

实现 Lambda 类（省去了注册和支持的代码）:

```
public static class Lambda extends Expr {
    // Tree element
    public List<LocalVarDef> params;
    public Expr expr;
    public Block block;

    public Lambda(Pos pos, List<LocalVarDef> params, Expr expr, Block block) {
        super(Kind.LAMBDA, "Lambda", pos);
        this.params = params;
        this.expr = expr;
        this.block = block;
    }

    @Override
    public Object treeElementAt(int index) {
        return switch (index) {
            case 0 -> params;
            case 1 -> (expr == null ? block : expr);
            default -> throw new IndexOutOfBoundsException(index);
        };
    }

    @Override
    public int treeArity() {
        return 2;
    }

    @Override
    public <C> void accept(Visitor<C> v, C ctx) {
        v.visitLambda(this, ctx);
    }
}
```

然后添加表达式:

```
    | FUN '(' VarList ')' ARROW Expr
      {
          $$ = svExpr(new Lambda($1.pos, $3.varList, $6.expr, null));
      }
    | FUN '(' VarList ')' Block
      {
          $$ = svExpr(new Lambda($1.pos, $3.varList, null, $5.block));
      }
```

3. 心得与困难：

（1）更加彻底的认识到添加新功能的常规操作：添加关键字-注册新类并支持-实现类定义-添加表达式

3°函数调用：

1. 任务要求与指导：

   由语法规范可知需要更改 Call 类的定义。

   函数调用

   原来只能调用成员方法和静态方法，现在可以调用任意类型为函数类型的表达式（其本质就是个函数）。

   语法规范：将原来的

   ```
   call ::= (expr '.')? id '(' exprList ')'
   ```

   变为

   ```
   call ::= expr '(' exprList ')'
   ```

2. 实验步骤：

修改 Call 定义，去除不需要的参数。

```java
public static class Call extends Expr {
    // Tree elements
    public Expr expr;
    public List<Expr> args;
    //
    public Call(Expr expr, List<Expr> args, Pos pos) {
        super(Kind.CALL, "Call", pos);
        this.expr = expr;
        this.args = args;
    }

    @Override
    public Object treeElementAt(int index) {
        return switch (index) {
        case 0 -> expr;
        case 1 -> args;
        default -> throw new IndexOutOfBoundsException(index);
        };
    }

    @Override
    public int treeArity() {
        return 2;
    }

    @Override
    public <C> void accept(Visitor<C> v, C ctx) {
        v.visitCall(this, ctx);
    }
}
```

修改表达式。

```
|   Expr '(' ExprList ')'
    {
        $$ = svExpr(new Call($1.expr, $3.exprList, $2.pos));
    }
;
```

3. 困难与心得:

（1）与同学商议之后决定删除不必要的参数，并在此基础上直接重写了整个 Call 类。在之后的修改表达式的过程中，又出现了需要更改全局所有调用了 Call 的构造函数的表达式的问题。

任务 4：习题

1. AST 结点间是有继承关系的。若结点 A 继承了 B，那么语法上会不会 A 和 B 有什么关系？限用 100 字符内一句话说明。

答：A的构造函数需要调用B的构造函数，继承关系表示A是特化后的B，A是B的一部分。

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。

答：由ElseClause的表达式可以看出来，它只会对else+stmt和无else的情况进行推导，有效避免了只有else而无实现的问题。

3. PA1-A 在概念上，如下图所示：

```
作为输入的程序（字符串）
    --> lexer --> 单词流 (token stream)
    --> parser --> 具体语法树 (CST)
    --> 一通操作 --> 抽象语法树 (AST)
```

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。
答：区别在于本框架中CST和AST同时构造，其中编译时产生错误时显示的yysv数组即为CST，而构造该数组需要产生AST上的节点，二者互相依赖最终同时产生。

任务5：总结

完成PA的过程让我意识到我对词法分析和文法分析的理解还不太透彻，在完成任务的过程中和同学不断讨论，加深了我对这两个过程的理解。jacc文件中表达式的构造让我觉得非常有意思，也是我第一次在代码中看到自动机形式的表达式，让我眼前一亮。实现过程中碰到的最大困难就是IDE自动补全时容易出现大小写错乱和少字被忽略的情况，导致gradle build的过程不断报错，不过在一次次解决的过程中也让我对整个结构的理解更为充分，总的来说感觉收益颇丰。