PA1-A实验报告

特性一

问题描述

用对象复制语句：

```
        scopy (id, expr) ;
```

来实现对象的浅复制。

实现方法：

- 添加关键词socpy
  - 在Lexer.l中添加scopy的识别规则
  - 在Parser.y中添加终结符SCOPY的定义
- 在Parser.y中添加对应的语法：

```
OCStmt          :   SCOPY '(' IDENTIFIER ',' Expr ')'
                    {
                        $$.stmt = new Tree.ScopyClass($3.ident,$5.expr,$1.loc);
                    }
                ;
```

- 在Tree.java中构建对应的语法类

```
    public static class ScopyClass extends Tree {
        public String identifier;
        public Expr expr;

        public ScopyClass(String ident, Expr expr,Location loc) {
            super(SCOPY, loc);
            this.identifier = ident;
            this.expr = expr;
        }

        @Override
        public void accept(Visitor v) {
            v.visitTree(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("scopy");
            pw.incIndent();
            pw.println(identifier);
            expr.printTo(pw);
            pw.decIndent();
        }
```

```
    }
```

特性二

问题描述

- 添加特性，使得用sealed描述的类无法被继承，语法形式如

```
    sealed class <identifier> extends <identifiers> {<Field*>}
```

实现方法

- 添加关键词sealed
  - 在Lexer.l中添加sealed的识别规则
  - 在Parser.y中添加终结符SEALED的定义
- 在Parser.y中添加对应的文法：

```
ClassDef        :   CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
                    {
                        $$.cdef = new Tree.ClassDef($2.ident, $3.ident, $5.flist, $1.loc);
                    }
                |   SEALED CLASS IDENTIFIER ExtendsClause '{' FieldList '}'
                    {
                        $$.cdef = new Tree.SealedClassDef($3.ident, $4.ident, $6.flist, $1.loc);
                    }
                ;
```

- 在Tree.java中,对相应语法节点类进行修改如下：

```
    public static class SealedClassDef extends ClassDef {

        public String name;
        public String parent;
        public List<Tree> fields;

        public SealedClassDef(String name, String parent, List<Tree> fields,
                Location loc) {
            //super(SEALEDCLASSDEF, loc);
            super(name,parent,fields,loc);
            this.name = name;
            this.fields = fields;
            this.parent = parent;
        }

        @Override
        public void accept(Visitor v) {
            v.visitSealedClassDef(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("sealed class " + name + " "
                + (parent != null ? parent : "<empty>"));
```

```
        }
    }
```

特性三

问题描述

```
        if{ <expr1> : <stmt1> |||
            <expr2> : <stmt2> |||
            ...
            <exprN> : <stmtN> }
```

实现方法

- 添加新的操作符 `|||` ，在Parser.y中添加识别表示符的规则，在SemValue.java中添加对应的case。
- 实现相应的文法，这里主要注意的点是要实现任意多项的串行条件语句分支，同时还要注意采用左递归，避免栈溢出：
  - `GuardedStmt : IF '{' IfBranches '}'`

    `;`
  - `IfSubStmt : Expr ':' Stmt`

    `;`
  - `IfBranches : IfBrancher GUARD IfSubStmt`

    `| IfSubStmt`

    `| /* empty */`

    `;`
- 在Tree.java中实现的对应的语法节点类为：

```
    public static class IfSubStmt extends Expr {
        public Expr condition;
        public Tree trueBranch;

        public IfSubStmt(Expr condition, Tree trueBranch,
                Location loc) {
            super(IFSUBSTMT, loc);
            this.condition = condition;
            this.trueBranch = trueBranch;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfSubStmt(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("guard");
            pw.incIndent();
```

```
            condition.printTo(pw);
            trueBranch.printTo(pw);
            pw.decIndent();
        }
    }

    public static class GuardedStmt extends Tree {
        public List<Expr> branches;

        public GuardedStmt(List<Expr> branches, Location loc) {
            super(IFBRANCHES,loc);
            this.branches = branches;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("guarded");
            pw.incIndent();
            if(branches!= null) {
                for(Expr substmt : branches) {
                    substmt.printTo(pw);
                }
            }else {
                pw.println("<empty>");
            }
            pw.decIndent();
        }
    }
```

特性四

问题描述

- 实现自动类型推导，比如下面的语句：

```
    var x = 1;
```

自动推导为int类型；

实现方法

- 添加新的关键词var到Lexer.l中
- 在Parser.y中：
  - 添加终结符VAR
  - 添加类型推导的对应文法：

    LValue : VAR IDENTIFIER

```
    | ...
```

- 在Tree.java中添加对应的语法节点类：

```
    public static class VarIdent extends LValue {

        public String name;
        public boolean isDefined;

        public VarIdent(String name, Location loc) {
            super(VARIDENT, loc);
            this.name = name;
        }

        @Override
        public void accept(Visitor v) {
            v.visitVarIdent(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("var " + name);
        }
    }
```

特性五

问题描述

支持若干与一维数组有关的表达式语句，具体包括：

- 数组常量，形如[c1,c2,c3,...,cn],其中c1,c2,c3,...,cn是同一类型的常量，n不小于0
- 数组初始化常量表达式，形如 E %% n 表示返回一个大小为n的数组常量，元素类型同表达式E的类型，每个元素的值被置为E的当前取值
- 数组拼接表达式，形如： E1 ++ E2，表示把两个同类型的数组E1和E2拼接成一个更长的数组
- 取子数组表达式，形如： E [E1:E2] 表示从数组E取出下标位于闭区间[E1:E2]的一段元素构成子数组，如果闭区间不合法，则返回空数组。
- 数组下标动态访问表达式，形如 E[E1]default E' 其中E1为整数类型表达式，E为数组类型表达式，E'为表达式,E'与E的元素具有相同类型。
- Python风格的数组comprehension表达式，形如 [E' for x in E if B]或者当B恒为true，简写为: [E' for x in E]
- 数组迭代语句，形如：foreach(var x in E)S 或 foreach(Type x in E)S

实现过程

- 数组常量表达式的实现：
    - 语法如下：
        -

```
ArrayConstant   :   '[' Constants ']'
                    {
                        $$.elist = $2.elist;
                        $$.expr = new Tree.ArrayConstant($2.elist,$1.loc);
                    }
                |   '[' ']'
                    {
                        $$.expr = new Tree.ArrayConstant(null,$1.loc);
                    }
                ;

Constants       :   Constant ',' Constants
                    {
                        $$.elist = $3.elist;
                        $$.elist.add(0,$1.expr);
                    }
                |   Constant
                    {
                        $$.elist = new ArrayList<Tree.Expr>();
                        $$.elist.add($1.expr);
                    }
                ;
```

- 对应的语法节点实现如下：

```java
public static class ArrayConstant extends Expr{
        public List<Expr> elist;

        public ArrayConstant(List<Expr> elist, Location loc) {
            super(ARRAYCONST,loc);
            this.elist = elist;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("array const");
            pw.incIndent();
            if(elist != null) {
                for(Expr e : elist) {
                    e.printTo(pw);
                }
            }else {
                pw.println("<empty>");
            }
            pw.decIndent();
        }
    }
```

        - 无需增加关键词
- 数组初始化常量表达式的实现：
    - 需要定义新的操作符%%以及++，在Lexer.l中添加相应定义：

```
    // 识别操作符的规则
"<="                { return operator(Parser.LESS_EQUAL);    }
">="                { return operator(Parser.GREATER_EQUAL);}
"=="                { return operator(Parser.EQUAL);         }
"!="                { return operator(Parser.NOT_EQUAL);     }
"&&"                { return operator(Parser.AND);           }
"||"                { return operator(Parser.OR);            }
"|||"               { return operator(Parser.GUARD);         }
"%%"                { return operator(Parser.MOMO);          }
"++"                { return operator(Parser.PLUSPLUS);      }
{SIMPLE_OPERATOR}   { return operator((int)yycharat(0));     }
```

      并在Parser.y中定义优先级：

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

    - 在Parser.y中实现对应的语法：

```
    |   Expr MOMO Expr
        {
            $$.expr = new Tree.ArrayRepeat($1.expr,$3.expr,$1.loc);
        }
```

    - 对应的语法节点为：

```
public static class ArrayRepeat extends Expr{
        public Expr expr;
        public Expr value;
```

```
        public ArrayRepeat(Expr expr, Expr value, Location loc) {
            super(ARRAYREPEAT,loc);
            this.expr = expr;
            this.value = value;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("array repeat");
            pw.incIndent();
            expr.printTo(pw);
            value.printTo(pw);
            pw.decIndent();
        }
    }
```

- 数组拼接表达式实现：
  - 对应的语法：

```
    Expr PLUSPLUS Expr
    {
        $$.expr = new Tree.PlusPlus($1.expr,$3.expr,$1.loc);
    }
```

  - 对应的语法节点：

```
public static class PlusPlus extends Expr{
        public Expr arr1;
        public Expr arr2;

        public PlusPlus(Expr arr1, Expr arr2, Location loc) {
            super(ARRAYREPEAT,loc);
            this.arr1 = arr1;
            this.arr2 = arr2;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("array concat");
            pw.incIndent();
            arr1.printTo(pw);
            arr2.printTo(pw);
```

```
            pw.decIndent();
        }
    }
```

- 取子数组表达式：
  - 对应语法

```
Expr '[' Expr ':' Expr ']'
{
    $$.expr = new Tree.ArrayRange($1.expr,$3.expr,$5.expr,$1.loc);
}
```

  - 对应语法节点：

```
    public static class ArrayRange extends Expr{
        public Expr arr;
        public Expr begin;
        public Expr end;
        public Location loc;

        public ArrayRange(Expr expr1, Expr expr2, Expr expr3, Location loc) {
            super(ARRAYRANGE,loc);
            this.arr = expr1;
            this.begin = expr2;
            this.end = expr3;
            this.loc = loc;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("arrref");
            pw.incIndent();
            arr.printTo(pw);
            pw.println("range");
            pw.incIndent();
            begin.printTo(pw);
            end.printTo(pw);
            pw.decIndent();
            pw.decIndent();
        }
    }
```

- 数组下标动态访问表达式：
  - 增加关键词default，在Lexer.l中添加识别的表达式，并在Parse.y中定义相应的终结符DEFAULT
  - 对应的语法为：
    - `Expr ::= Expr [ Expr ] default Expr | ...`

- 对应的语法节点：

```
    public static class ArrayDefault extends Expr{
        public Expr arr;
        public Expr idx;
        public Expr defaultIdx;
        public Location loc;

        public ArrayDefault(Expr expr1, Expr expr2, Expr expr3, Location loc) {
            super(ARRAYRANGE,loc);
            this.arr = expr1;
            this.idx = expr2;
            this.defaultIdx = expr3;
            this.loc = loc;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("arrref");
            pw.incIndent();
            arr.printTo(pw);
            idx.printTo(pw);
            pw.println("default");
            pw.incIndent();
            defaultIdx.printTo(pw);
            pw.decIndent();
            pw.decIndent();
        }
    }
```

- Python风格的数组comprehension表达式：
  - 新增关键词in，在Lexer.l中添加识别规则，在Parser.y中添加终结符IN
  - 对应的语法：

    Expr : [Expr FOR IDENTIFIER IN Expr]

    | [Expr FOR IDENTIFIER IN Expr IF Expr]
  - 对应的语法节点：

```
public static class ArrayComp extends Expr{
        public String ident;
        public Expr todo;
        public Expr arr;
        public Expr judge;
        public Location loc;

        public ArrayComp(Expr todo, String ident, Expr arr, Expr judge, Location loc) {
```

```
            super(ARRAYCOMP,loc);
            this.todo = todo;
            this.ident = ident;
            this.arr = arr;
            this.judge = judge;
            this.loc = loc;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("array comp");
            pw.incIndent();
            pw.println("varbind " + ident);
            arr.printTo(pw);
            if(judge != null)
                judge.printTo(pw);
            else
                pw.println("boolconst true");
            todo.printTo(pw);
            pw.decIndent();
        }
    }
```

- 数组迭代语句：
  - 新增关键词foreach，在Lexer.l中添加识别规则，在Parser.y中添加终结符FOREACH
  - 对应的语法：
    - `Stmt : ForeachStmt | ...`
    - `ForeachStmt : FOREACH '(' VAR IDENTIFIER IN Expr WHILE Expr ')' StmtBlock`

      `| FOREACH '(' VAR IDENTIFIER IN Expr')' StmtBlock`

      `| FOREACH '(' Type IDENTIFIER IN Expr WHILE Expr ')' StmtBlock`

      `| FOREACH '(' Type IDENTIFIER IN Expr ')' StmtBlock`
  - 对应的语法节点：

```
    public static class ForeachStmt extends Tree{
        public TypeLiteral type;
        public String ident;
        public Expr arr;
        public Expr judge;
        public Tree stmtBlock;
        public Location loc;

        public ForeachStmt(TypeLiteral type, String ident, Expr arr, Expr judge, Tree
stmtBlock, Location loc) {
            super(FOREACHSTMT,loc);
```

```
            this.type = type;
            this.ident = ident;
            this.arr = arr;
            this.judge = judge;
            this.stmtBlock = stmtBlock;
            this.loc = loc;
        }

        @Override
        public void accept(Visitor v) {
            v.visitIfBranches(this);
        }

        @Override
        public void printTo(IndentPrintWriter pw) {
            pw.println("foreach");
            pw.incIndent();
            if(type == null)
                pw.println("varbind " + ident + " var");
            else {
                pw.print("varbind " + ident + " ");
                type.printTo(pw);
                pw.println();
            }
            arr.printTo(pw);
            if(judge != null)
                judge.printTo(pw);
            else
                pw.println("boolconst true");
            stmtBlock.printTo(pw);
            pw.decIndent();
        }
    }
```