编译原理PA2实验报告

张钰晖 计55 2015011372 yuhui-zh15@mails.tsinghua.edu.cn

任务描述

PA1阶段，我们完成了**词法分析**、**语法分析**，生成了**抽象语法树(AST)**。

PA2阶段，我们要基于PA1的抽象语法树，实现**构造符号表**、**静态语义检查**。

文件说明

在本阶段，以下文件非常重要，主要需要修改以下文件。

文件名	含义	说明
typecheck/BuildSym	第一趟扫描，建立符号表	增加新的语法特性，建立符号表
typecheck/TypeCheck	第二趟扫描，进行语义检查	增加新的语法特性，对不合法的情况进行报错
frontend/*	编译器最前端	拷贝PA1A文件
type/*	类型定义	增加新的类型
tree/*	抽象语法树的各种节点	沿用PA1A修改部分
error/*	编译错误的类	增加新的错误类

总体来看，本阶段修改如下：

- typecheck
  - 修改BuildSym类
  - 修改TypeCheck类（主要修改）
- frontend
  - 修改沿用PA1A
- type
  - 修改BaseType类
- tree
  - 修改Tree类
- error
  - 新增SuperInStaticFuncError类
  - 新增BadPrintCompArgError类
  - 新增FieldNotSupportedError类

- 新增NoParentClassError类
- 新增BadCopyArgError类
- 新增IncompatCopyAssignError类
- 新增IncompatCaseExprError类
- 新增DuplicateConditionError类
- 新增DifferentCaseTypeError类
- 新增BadDoStmtArgError类

实验内容及实现

本次实验分为6个阶段，复用PA1A结果及实现5个新文法特性，主要修改列在下方。

步骤零：复用结果

本阶段需要复用PA1A词法分析和语法分析结果，实现以下新的文法特性：

- 整复数类型的支持：本阶段需要增加整复数类型，增加识别复数常量虚部功能，增加获取复数实部、虚部及强制转换复数表达式，增加复数打印语句。
- case表达式的支持：本阶段需要支持case表达式，语法为case(表达式) {常量1:表达式1,..., default:表达式}。
- super表达式的支持：本阶段需要支持super表达式，类似this表达式。
- 对象复制的支持：本阶段需要支持深复制dcopy()和浅复制scopy()表达式。
- 串行循环卫士的支持：本阶段需要支持串行循环卫士语句，语法为do E1:S1|||E2:S2... od。

本阶段修改文件如下：

文件名	修改
BaseLexer.java	复用修改
BaseParser.java	复用修改
Lexer.l	复用修改
Parser.y	复用修改
SemValue.java	复用修改
Tree.java	沿用修改

> 注：由于笔者实现PA1理解不够深入，实现case表达式和穿行循环卫士时未考虑继承的一些问题，对部分少许代码进行了修改与重构。

步骤一：支持整复数类型

1. 增加整复数类型和常量虚部

- 修改类type/BaseType.java

增加基本类型COMPLEX。

```
public class BaseType extends Type {
    ...
    public static final BaseType COMPLEX = new BaseType("complex");
    ...
}
```

- 修改类typecheck/BuildSym.java

修改函数visitTypeIdent新增COMPLEX。

```
@Override
public void visitTypeIdent(Tree.TypeIdent type) {
    switch (type.typeTag) {
    ...
    case Tree.COMPLEX:
        type.type = BaseType.COMPLEX;
        break;
    ...
    }
}
```

- 修改类typecheck/TypeCheck.java

修改函数visitTypeIdent新增COMPLEX。

```
@Override
public void visitTypeIdent(Tree.TypeIdent type) {
    switch (type.typeTag) {
    ...
    case Tree.COMPLEX:
        type.type = BaseType.COMPLEX;
        break;
    ...
    }
}
```

修改函数visitLiteral新增IMG。

```
@Override
public void visitLiteral(Tree.Literal literal) {
    switch (literal.typeTag) {
    ...
    case Tree.IMG:
        literal.type = BaseType.COMPLEX;
        break;
    ...
    }
}
```

2. 对于新增表达式 @E 和 $E，E必须是complex类型的表达式，且这两个表达式计算结果的类型为int。对于新增表达式 #E，E必须是int类型的表达式，且该表达式计算结果的类型为complex。

- 修改类typecheck/TypeCheck.java

修改函数visitUnary，处理@、$、#一元运算符。

```
@Override
public void visitUnary(Tree.Unary expr) {
    ...
    if (expr.expr.type.equal(BaseType.ERROR)) {
        expr.type = BaseType.ERROR;
    }
    else if (expr.tag == Tree.NEG) {
        if (expr.expr.type.equal(BaseType.INT)) {
            expr.type = BaseType.INT;
        } else {
            issueError(new IncompatUnOpError(expr.getLocation(), "-",
                    expr.expr.type.toString()));
            expr.type = BaseType.ERROR;
        }
    }
    else if (expr.tag == Tree.NOT) {
        if (expr.expr.type.equal(BaseType.BOOL)) {
            expr.type = BaseType.BOOL;
        } else {
            issueError(new IncompatUnOpError(expr.getLocation(), "!",
                    expr.expr.type.toString()));
            expr.type = BaseType.ERROR;
        }
    }
    else if (expr.tag == Tree.GETREAL) {
        if (expr.expr.type.equal(BaseType.COMPLEX)) {
            expr.type = BaseType.INT;
        } else {
            issueError(new IncompatUnOpError(expr.getLocation(), "@",
                    expr.expr.type.toString()));
            expr.type = BaseType.ERROR;
        }
    }
    else if (expr.tag == Tree.GETIMG) {
        if (expr.expr.type.equal(BaseType.COMPLEX)) {
            expr.type = BaseType.INT;
        } else {
            issueError(new IncompatUnOpError(expr.getLocation(), "$",
                    expr.expr.type.toString()));
            expr.type = BaseType.ERROR;
        }
    }
    else if (expr.tag == Tree.TOCOMPLEX) {
```

```
        if (expr.expr.type.equal(BaseType.INT)) {
            expr.type = BaseType.COMPLEX;
        } else {
            issueError(new IncompatUnOpError(expr.getLocation(), "#",
                    expr.expr.type.toString()));
            expr.type = BaseType.ERROR;
        }
    }
}
```

3. 本学期，我们限定复数表达式仅包含加法（+）和乘法（*）运算，即不支持含有其他运算的表达式。为符合习惯，我们允许复数和整数之间进行混合运算，运算结果仍为复数类型。然而，为了后续中间代码生成（见PA3实验）的方便，我们建议大家可以在本次实验中将混合运算中的整数自动转化为复数（插入一个#运算的结点），这样可以简化存储分配方案（以浪费一些存储为代价），从而简化PA3的工作。当然，PA2的测例不会评估你是否做了这件事。 关于运算符的错误处理，遇见不合法的运算报错返回ERROR。操作数中存在ERROR不报错返回ERROR。例如string + string 报错并返回ERROR。-(ERROR) 不报错并返回ERROR。

- 修改类typecheck/TypeCheck.java

修改函数checkBinaryOp，处理+、*、=、≠二元运算符。

```
private Type checkBinaryOp(Tree.Expr left, Tree.Expr right, int op,
Location location) {
    ...
    if (left.type.equal(BaseType.ERROR) ||
right.type.equal(BaseType.ERROR)) {
        return BaseType.ERROR;
    }
    ...
    switch (op) {
    case Tree.PLUS:
    case Tree.MUL:
        if (left.type.equal(BaseType.COMPLEX) &&
right.type.equal(BaseType.INT)) {
            right = new Tree.Unary(Tree.TOCOMPLEX, right, right.loc);
            right.accept(this);
            compatible = true;
            returnType = BaseType.COMPLEX;
        }
        else if (left.type.equal(BaseType.INT) &&
right.type.equal(BaseType.COMPLEX)) {
            left = new Tree.Unary(Tree.TOCOMPLEX, left, left.loc);
            left.accept(this);
            compatible = true;
            returnType = BaseType.COMPLEX;
        }
        else {
            compatible = (left.type.equals(BaseType.INT) ||
left.type.equals(BaseType.COMPLEX))
```

```
                    && left.type.equal(right.type);
            returnType = left.type;
        }
        break;
    case Tree.MINUS:
    case Tree.DIV:
        compatible = left.type.equals(BaseType.INT)
                && left.type.equal(right.type);
        returnType = left.type;
        break;
    ...
    case Tree.EQ:
    case Tree.NE:
        if (left.type.equal(BaseType.COMPLEX) &&
right.type.equal(BaseType.COMPLEX)) {
            compatible = false;
            returnType = BaseType.ERROR;
        }
        else {
            compatible = left.type.compatible(right.type)
                    || right.type.compatible(left.type);
            returnType = BaseType.BOOL;
        }
        break;
    ...
    }
    if (!compatible) {
        ...
        returnType = BaseType.ERROR;
    }
    ...
}
```

4. 新增语句复数打印语句PrintComp（E1，E2，...，En）表示复数表达式E1，E2，...，En计算结果的显示，其参数表达式要求具有complex类型。

- 增加类error/BadPrintCompArgError.java

新增错误BadPrintCompArgError，当PrintComp参数非complex类型时报错。

```
public class BadPrintCompArgError extends DecafError {
    private String count;
    private String type;
    public BadPrintCompArgError(Location location, String count, String
type) {
        super(location);
        this.count = count;
        this.type = type;
    }
    @Override
    protected String getErrMsg() {
        return "incompatible argument " + count + ": " + type + " given,
complex expected";
    }
}
```

- 修改类typecheck/TypeCheck.java

新增函数visitPrintComp，用于打印复数。

```
@Override
public void visitPrintComp(Tree.PrintComp printCompStmt) {
    int i = 0;
    for (Tree.Expr e : printCompStmt.exprs) {
        e.accept(this);
        i++;
        if (!e.type.equal(BaseType.ERROR) &&
!e.type.equal(BaseType.COMPLEX)) {
            issueError(new BadPrintCompArgError(e.getLocation(), Integer
                    .toString(i), e.type.toString()));
        }
    }
}
```

5. 若不符合上述情形，则报告相应的语义错误。

阶段二：支持Case表达式

1. E必须是int类型的表达式。

- 增加类error/IncompatCaseExprError.java

新增错误IncompatCaseExprError，当Case表达式非int类型时报错。

```
public class IncompatCaseExprError extends DecafError {
    private String type;
    public IncompatCaseExprError(Location location, String type) {
        super(location);
        this.type = type;
    }
    @Override
    protected String getErrMsg() {
        return "incompatible case expr: " + type + " given, but int
expected";
    }
}
```

2. C1、C2、...、Cn是互不相同的int类型常量。

- 增加类error/DuplicateConditionError.java

新增错误DuplicateConditionError，当C1、C2、...、Cn重复时报错。

```
public class DuplicateConditionError extends DecafError {
    public DuplicateConditionError(Location location) {
        super(location);
    }
    @Override
    protected String getErrMsg() {
        return "condition is not unique";
    }
}
```

3. E1、E2、...、En是和En+1具有相同类型的表达式。

- 增加类error/DifferentCaseTypeError.java

新增错误DifferentCaseTypeError，当E1、E2、...、En和En+1类型不同时报错。

```
public class DifferentCaseTypeError extends DecafError {
    private String casetype;
    private String defaulttype;
    public DifferentCaseTypeError(Location location, String casetype,
String defaulttype) {
        super(location);
        this.casetype = casetype;
        this.defaulttype = defaulttype;
    }
    @Override
    protected String getErrMsg() {
        return "type: " + casetype + " is different with other expr's
type " + defaulttype;
    }
}
```

4. 该表达式计算结果的类型与E1、E2、...、En 以及 En+1 具有相同的类型。

- 修改类typecheck/TypeCheck.java

修改函数visitCaseExpr，进行语义检查。

```
@Override
public void visitCaseExpr(Tree.CaseExpr caseExpr) {
    caseExpr.conditionexpr.accept(this);
    caseExpr.defaultexpr.accept(this);
    for (Tree.ACaseExpr expr: caseExpr.casesexpr) expr.accept(this);
    if (!caseExpr.conditionexpr.type.equal(BaseType.INT) &&
!caseExpr.conditionexpr.type.equal(BaseType.ERROR)) {
        issueError(new IncompatCaseExprError(caseExpr.conditionexpr.loc,
caseExpr.conditionexpr.type.toString()));
    }
    Set<Integer> casesnum = new HashSet<>();
    caseExpr.type = caseExpr.defaultexpr.type;
    for (Tree.ACaseExpr expr: caseExpr.casesexpr) {
        if (casesnum.contains((Integer)expr.literal.value)) {
            issueError(new DuplicateConditionError(expr.literal.loc));
        }
        else {
            casesnum.add((Integer)expr.literal.value);
        }
        if (!caseExpr.defaultexpr.type.equal(BaseType.ERROR) &&
!expr.expr.type.equal(caseExpr.defaultexpr.type)) {
            issueError(new DifferentCaseTypeError(expr.loc,
expr.expr.type.toString(), caseExpr.defaultexpr.type.toString()));
            caseExpr.type = BaseType.ERROR;
        }
    }
}
```

修改函数visitACaseExpr，进行语义检查。

```
@Override
public void visitACaseExpr(Tree.ACaseExpr acaseExpr) {
    acaseExpr.literal.accept(this);
    acaseExpr.expr.accept(this);
    if(!acaseExpr.literal.type.equal(BaseType.INT) &&
!acaseExpr.literal.type.equal(BaseType.ERROR)) {
        issueError(new IncompatCaseExprError(acaseExpr.literal.loc,
acaseExpr.literal.type.toString()));
    }
}
```

5. 若不符合上述情形，则报告相应的语义错误。

阶段三：支持Super表达式

1. 类似于this表达式，super表达式返回当前对象，其类型为当前对象的class。

- 修改类typecheck/TypeCheck.java

新增函数visitSuperExpr，支持super表达式。

```
@Override
public void visitSuperExpr(Tree.SuperExpr superExpr) {
    if (currentFunction.isStatik()) {
        issueError(new SuperInStaticFuncError(superExpr.getLocation()));
        superExpr.type = BaseType.ERROR;
    } else {
        superExpr.type = ((ClassScope)
table.lookForScope(Scope.Kind.CLASS)).getOwner().getType();
    }
}
```

2. 本次实验中，我们限定仅支持面向super的函数调用（call表达式），而不支持面向super的成员变量访问。

- 增加类error/FieldNotSupportedError.java

新增错误FieldNotSupportedError，当super访问成员变量时报错。

```
public class FieldNotSupportedError extends DecafError {
    public FieldNotSupportedError(Location location) {
        super(location);
    }
    @Override
    protected String getErrMsg() {
        return "super.member_var is not supported";
    }
}
```

- 修改类typecheck/TypeCheck.java

修改visitIdent函数，当super访问成员变量时报错。

```
@Override
public void visitIdent(Tree.Ident ident) {
    if (ident.owner == null) {
        ...
    } else {
        ...
        if (!ident.owner.type.equal(BaseType.ERROR)) {
            if (ident.owner instanceof Tree.SuperExpr) {
                issueError(new
FieldNotSupportedError(ident.getLocation()));
                ident.type = BaseType.ERROR;
            }
            else if (ident.owner.isClass ||
!ident.owner.type.isClassType()) {
                ...
            }
            ...
        }
    }
}
```

3. 面向super的函数调用（call表达式）super.f(...)，是调用当前对象的class的超类中的成员函数，即从其父类开始向上搜索类层次首次发现的成员函数f(...)，如果未搜索到则报错。

- 增加类error/NoParentClassError.java

新增错误NoParentClassError，当未搜索到父类时报错。

```
public class NoParentClassError extends DecafError {
    private String name;
    public NoParentClassError(Location location, String name) {
        super(location);
        this.name = name;
    }
    @Override
    protected String getErrMsg() {
        return "no parent class exist for " + name;
    }
}
```

4. super运算不能出现在static的函数中。

- 增加类error/SuperInStaticFuncError.java

新增错误SuperInStaticFuncError，当super出现在static函数时报错。

```
public class SuperInStaticFuncError extends DecafError {
    public SuperInStaticFuncError(Location location) {
        super(location);
    }
    @Override
    protected String getErrMsg() {
        return "can not use super in static function";
    }
}
```

5. super只有在函数调用时才寻找父类，其他时候和this表达式行为类似。

- 修改类typecheck/TypeCheck.java

修改函数checkCallExpr和visitCallExpr，对super进行语义检查。

```
@Override
private void checkCallExpr(Tree.CallExpr callExpr, Type receiverType,
Symbol f) {
    // Type receiverType = callExpr.receiver == null ? ((ClassScope)
table.lookForScope(Scope.Kind.CLASS)).getOwner().getType():
callExpr.receiver.type;
    ...
}
```

```java
@Override
public void visitCallExpr(Tree.CallExpr callExpr) {
    if (callExpr.receiver == null) {
        ...
        Type receiverType = ((ClassScope)table.lookForScope(Scope.Kind.CLASS)).getOwner().getType();
        checkCallExpr(callExpr, receiverType, cs.lookupVisible(callExpr.method));
        ...
    }
    ...
    if (callExpr.receiver instanceof Tree.SuperExpr) {
        if (((ClassType)callExpr.receiver.type).getParentType() == null) {
            issueError(new NoParentClassError(callExpr.getLocation(), callExpr.receiver.type.toString()));
            callExpr.type = BaseType.ERROR;
        }
        else {
            Symbol f = null;
            ClassType currentType = ((ClassType)callExpr.receiver.type).getParentType();
            while (currentType != null && (f = currentType.getClassScope().lookupVisible(callExpr.method)) == null) {
                currentType = currentType.getParentType();
            }
            if (f == null) {
                checkCallExpr(callExpr, ((ClassType)callExpr.receiver.type).getParentType(), f);
            }
            else {
                checkCallExpr(callExpr, currentType, f);
            }
        }
        return;
    }
    ClassScope cs = ((ClassType) callExpr.receiver.type).getClassScope();
    Type receiverType = callExpr.receiver.type;
    checkCallExpr(callExpr, receiverType, cs.lookupVisible(callExpr.method));
}
```

6. 若不符合上述情形，则报告相应的语义错误。

阶段四：支持对象复制

1. 对于新增表达式深复制 dcopy(e) 和浅复制 scopy(e)，e必须是class类型的表达式，且这两个表达式计算结果将生成新的对象，该对象的类型为与e相同的class类型。

- 增加类error/BadCopyArgError.java

新增错误BadCopyArgError，当对象类型非class时报错。

```
public class BadCopyArgError extends DecafError {
    private String type;
    public BadCopyArgError(Location location, String type) {
        super(location);
        this.type = type;
    }
    @Override
    protected String getErrMsg() {
        return "expected class type for copy expr but " + type + "
given";
    }
}
```

- 修改类typecheck/Typecheck.java

修改函数visitDcopyExpr和visitScopyExpr，实现dcopy和scopy语义检查。

```
@Override
public void visitDcopyExpr(Tree.DcopyExpr dcopyExpr) {
    dcopyExpr.expr.accept(this);
    if (!dcopyExpr.expr.type.isClassType() &&
!dcopyExpr.expr.type.equal(BaseType.ERROR)) {
        issueError(new BadCopyArgError(dcopyExpr.loc,
dcopyExpr.expr.type.toString()));
        dcopyExpr.type = BaseType.ERROR;
    }
    else {
        dcopyExpr.type = dcopyExpr.expr.type;
    }
}
```

```
@Override
public void visitScopyExpr(Tree.ScopyExpr scopyExpr) {
    scopyExpr.expr.accept(this);
    if (!scopyExpr.expr.type.isClassType() &&
!scopyExpr.expr.type.equal(BaseType.ERROR)) {
        issueError(new BadCopyArgError(scopyExpr.loc,
scopyExpr.expr.type.toString()));
        scopyExpr.type = BaseType.ERROR;
    }
    else {
        scopyExpr.type = scopyExpr.expr.type;
    }
}
```

2. 当表达式dcopy(e) 和scopy(e) 出现在赋值语句中时，比如x=dcopy(e)、x=scopy(e)，被复制的变量x须具有与e相同的class类型。其他情况等同于普通表达式。

- 增加类error/IncompatCopyAssignError.java

新增错误IncompatCopyAssignError，当复制的变量与被复制变量的class类型不同时报错。

```
public class IncompatCopyAssignError extends DecafError {
    private String left;
    private String right;
    public IncompatCopyAssignError(Location location, String left,
String right) {
        super(location);
        this.left = left;
        this.right = right;
    }
    @Override
    protected String getErrMsg() {
        return "For copy expr, the source " + right + " and the
destination " + left + " are not same";
    }
}
```

- 修改类typecheck/Typecheck.java

修改函数visitAssign，当复制的变量与被复制变量的class类型不同时报错。

```
@Override
public void visitAssign(Tree.Assign assign) {
    ...
    if (assign.expr instanceof Tree.DcopyExpr || assign.expr instanceof
Tree.ScopyExpr) {
        if (!assign.left.type.equal(BaseType.ERROR) &&
!assign.expr.type.equal(BaseType.ERROR) &&
!assign.left.type.equal(assign.expr.type)) {
            issueError(new IncompatCopyAssignError(assign.getLocation(),
assign.left.type.toString(), assign.expr.type.toString()));
        }
    }
    else if (!assign.left.type.equal(BaseType.ERROR) &&
(assign.left.type.isFuncType() ||
!assign.expr.type.compatible(assign.left.type))) {
        ...
    }
}
```

3. 若不符合上述情形，则报告相应的语义错误。

阶段五：支持串行循环卫士

1. E1，E2，...,En 必须是bool类型的表达式。

- 增加类error/BadDoStmtArgError.java

新增错误BadDoStmtArgError，当E类型不为bool时报错。

```
public class BadDoStmtArgError extends DecafError {
    private String type;
    public BadDoStmtArgError(Location location, String type) {
        super(location);
        this.type = type;
    }
    @Override
    protected String getErrMsg() {
        return "The condition of Do Stmt requestd type bool but " + type
+ " given";
    }
}
```

2. 其静态语义规则类似于框架中已有的其他循环语句的规则。

- 修改类typecheck/BuildSym.java

新增函数visitDoStmt，构建Sym表。

```
@Override
public void visitDoStmt(Tree.DoStmt doStmt) {
    for (Tree.DoSubStmt doSubStmt: doStmt.dslist) {
        doSubStmt.stmt.accept(this);
    }
}
```

3. 支持break语句。

- 修改类typecheck/TypeCheck.java

新增函数visitDoStmt和visitDoSubStmt，支持break语句。

```
@Override
public void visitDoStmt(Tree.DoStmt doStmt) {
    breaks.add(doStmt);
    for (Tree.DoSubStmt doSubStmt: doStmt.dslist) {
        doSubStmt.accept(this);
    }
    breaks.pop();
}
```

```
@Override
public void visitDoSubStmt(Tree.DoSubStmt doSubStmt) {
    doSubStmt.expr.accept(this);
    doSubStmt.stmt.accept(this);
    if (!doSubStmt.expr.type.equal(BaseType.BOOL) &&
!doSubStmt.expr.type.equal(BaseType.ERROR)) {
        issueError(new BadDoStmtArgError(doSubStmt.expr.loc,
doSubStmt.expr.type.toString()));
    }
}
```

4. 语义错误的处理可可参照框架中已有的其他循环语句的处理方法。

技巧心得

本次作业难度相对较大，通过以下方法可以加速编程。

1. 仔细阅读测试样例及正确输出

尽管正确分析的流程应该是根据Decaf语言规范，对于每一种语句，找出其应该遵循的规则，针对违反规则的情况进行报错。

但由于所有错误都体现在了测试样例中，当充分理解测试样例的报错后，便可以较快的理解本次任务，从而在对应代码部分进行修改。

2. 评测脚本无法运行

将runAll.py中open(filename, 'a+')改为open(filename, 'r')即可解决。

总结

本次实验PA2相对前两次实验PA1难度提升明显，一个明显的特点就是实现代码量大，需要改动的地方较多，尽管实现的过程很痛苦，但在实现的过程中，充分的锻炼了笔者的编程能力，并且第一次理解和尝试了visitor设计模式，对语义分析的相关概念有了质的提高，也了解了符号表属性文法等知识，笔者在实践之中真正感受到了编译的神奇之处。

在实现PA2过程中，由于开始做PA1时理解不够深入，有些地方实现存在瑕疵，不得不返工重构部分PA1代码，不过在重构过程中笔者也对编译器的工作过程认识进一步加深。