Decaf PA2 说明

任务描述

经过了 PA1 的热身，相信大家已经对编译器的构造过程有了第一眼的印象。大家通过 PA1 的工作应该已经体会到使用自动构造工具编写编译器的便利(请考虑自己手工编写一系列高效率的词法和语法函数所需要的工作量)。在 PA2 中，我们将正式开始 Decaf 编译器的第一项关键工作——语义分析。

语义分析的目的是理解输入程序的含义，即弄清楚输入中所出现的每个标识符所指的内容以及明确输入中所出现的各种语句的含义。

对于标识符所指的内容，我们往往借助一种叫做“符号表”的数据结构来进行处理：随着语义分析过程的进行，当遇到符号定义的时候，我们需要往符号表中加入适当的记录信息；当遇到对符号的引用的时候，我们在符号表中查找这个符号的名字，并确定这个引用所指的是什么符号。如果引用一个没有定义的符号的话，我们将没法知道这个引用的具体含义，后面的翻译阶段也没法正常工作，因此需要把不合法的符号引用筛选出来。同样，如果有两个同样名字的符号在同一个作用域里面被定义，那么在引用这个名字的时候我们将不知道引用的到底是哪一个符号，因此这样的符号声明冲突也需要筛选出来。

对于语句的含义，我们需要根据 Decaf 语言规范中的规定对每一个语句或者表达式进行检查，看其是否合法。由于不符合语言规范的语句或者表达式所导致的执行效果是没有定义的，因此如果我们不把这样的不合法语句和表达式排除掉的话，那么在后面的翻译过程中编译器将不知道应该把它们翻译成什么。一般情况下，编译器对于语句和表达式只做类型检查，即检查这些语句和表达式的各个参数的数据类型是否符合规定，并且推断出它们执行结果的数据类型。

在 PA2 中，我们把语义分析分为对抽象语法树（下称 AST）的两趟扫描进行：第一趟扫描的时候建立符号表的信息，并且检测符号声明冲突以及跟声明有关的符号引用问题（例如 A 继承于 B，但是 B 没有定义的情况）；第二趟扫描的时候检查所有的语句以及表达式的参数的数据类型。在 PA2 中我们将借助 AST 结点的附加属性信息来进行语义分析。

希望大家通过 PA2，掌握类型系统的实现方法、作用域分析和符号表的构造方法，并能够把属性文法和语法制导的理论应用到实践中。

截止时间

PA2 持续时间为 18 天，提交时间以网络学堂为准。

本阶段的测试要求仍然是输出与标准输出完全一致（忽略空白字符和换行符）。在 PA2 中我们仍然会保留一部分不公开的测试例子(包括正例和反例，每个例子仅含有一种情况)，因此你需要在调试的时候自己设计一些例子进行测试，看看是否对所有的符号声明和引用都进行了检测，是否所有的语句和表达式都正确地进行了类型检查。

本阶段涉及的类和工具的说明

本阶段主要涉及的类和文件如下：

文件/类	含义	说明
typecheck/BuildSym	第一趟扫描，建立符号表	框架中的实现代码基本上够用了，对于个别新增语法可根据自己的需要进行适当的增加/修改
typecheck/TypeCheck	第二趟扫描，进行语义检查	你要在此文件中对相应的 ast 节点进行处理，主要工作是对不合法情况进行报错处理。
frontend/*	编译器最前端	第一阶段工作，你需要将第一阶段工作移植到本阶段中。
scope/*	作用域定义	不需要修改，请事先阅读
symbol/*	符号定义	不需要修改，请事先阅读
type/*	类型定义	不需要修改，请事先阅读
tree/*	抽象语法树的各种结点	第一阶段工作，你需要将第一阶段工作移植到本阶段中。
error/*	表示编译错误的类	依据测试样例，加入新错误类型。
Driver	Decaf 编译器入口	调试时可以修改
Option	编译器选项	不要修改
Location	文法符号的位置	不要修改
utils/*	辅助的工具类	可以增加，不要修改原来的部分
build.xml	Ant Build File	不要修改

PA2 阶段公开的测试例子放在 TestCases 的 S2 目录中，对于反例，我们在打印报错信息以后结束编译过程；对于正例，我们将把符号表的样子打印出来。

经过 PA1 的热身，大家应该对框架已经有了一定的了解，因此这个阶段我们不再提供单独的调试函数，请大家自行想办法进行调试。

特别说明

在本次实验中，依旧给出满足《decaf 语言规范》的基本框架实现，要求和第一阶段一样，增加新语言特性的实现，构建符号表，完成相应的语义分析以及错误处理。增加的语言特性包括：

1. 多行注释：Decaf 语言规范中只实现了单行注释，要求加入多行注释的支持（以“/*”开始，以“*/”结束）。
2. ++，--运算符：实现自增、自减单操作算子，形如 i++,++i,i--,--i。
   操作数：自增，自减运算符对应的操作数均为 Lvalue；
   优先级：i++优先级 高于 其他单操作算子 高于 ++i；
   操作过程：i++，先返回 i 的值，再进行自增；++i 先自增，再返回 i。

注：自增/自减的操作数在第一阶段设为普通表达式，而本阶段应通过静态检查限定其只能是 Lvalue 表达式。如，遇到表达式 1++时，应该报语义错误。

3. 三元运算符：实现三操作数算子？：，形如 A ？ B ： C。
   操作数：A 为条件表达式，B 和 C 类型要求一致；
   优先级和结合性：右结合，优先级 低于 || 高于 =；
4. switch-case 语句：实现 switch-case 控制结构，形如：

```
SwitchStmt    ::= switch ( Expr ) { CaseStmt* <DefaultStmt> }
CaseStmt      ::= case Constant : Stmt*
DefaultStmt   ::= default : Stmt*
```

5. repeat-until 循环结构：实现 repeat 循环结构，语法形如：

```
RepeatStmt    ::= repeat Stmt until ( BoolExpr) ;
```

具体要求和阶段一是类似的，学习框架中语义分析的实现，完成新特性的内容。

语义分析

语义分析，顾名思义是执行一系列关于程序语义的处理，并明确程序的含义，例如把引用到的标识符和具体的符号关联、分辨所使用的语言规则等等。语义分析的目标是尽可能深入地了解源程序的语义，并利用这些信息进行后面的工作（例如优化）。语义分析一般沿着语法树结构进行，并借助属性文法来开展，这种由语法树结构指导处理的做法称为“语法驱动”。所谓属性文法，就是给每个语法符号绑定一系列的属性后形成的扩展文法，这种文法有利于进行语法驱动的语义分析。关于属性文法的具体说明请参考本文档后面的部分。

语法驱动的语义分析过程是以遍历语法树的方式进行的，有两种主要的遍历方式：

1、语义分析和建立语法树同时进行，并且分析顺序与语法树建立顺序一致。这种方式的优点是编写的总代码量比较少，分析速度快，而且语法分析、语义分析和中间代码翻译是同时进行的；但是它的缺点在于语义分析能够利用的信息严重受限于语法分析的顺序，通常要求所引用的所有符号都必须在前面已经声明，而且实现的时候逻辑容易混乱。采用这种方式进行语义分析的话通常都只语法符号的综合属性，属性传递方向都是自下而上。

2、首先建立 AST，然后再通过一次或多次的遍历 AST（可能有回溯）来进行语义值传递、语义错误检测等。这种方式的优点是功能强大，能够尽最大可能提取源程序的语义信息，而且其逻辑简单，遍历语法树的代码不需要额外的专业基础也能轻易编写，因此被不少现代的编译器例如 GCC 等采用。但它也有明显的缺点：编码量相对较大，而且由于采用多趟遍历语法树，编译过程的时间开销也随之增大，然而这些缺点在现在一般不是大家重点关心的地方。另外，采用这种方式进行语义分析的时候往往同时采用继承属性和综合属性。

显然 Decaf 采用是后面一种方式（PA1 的工作就是建立 AST）。在 PA2 中，我们需要在 PA1 所建立的 AST 上进行多次遍历，借助关联在 AST 结点的各种属性（主要用综合属性）进行语义分析。

属性文法

所谓属性文法，即“带属性的文法”。属性文法是指这样扩展以后的上下文无关文法：

为每种语法符号关联一个可能为空的属性集合。这样做的意义在于通过这些属性，我们可以在遍历语法树的时候在某些结点中记录下一些计算结果，供子结点或者父结点的计算过程使用，甚至还可以供以后再次经过该结点的计算过程使用。由于语法树结点和语法符号是相对应的，下面为了便于理解起见，也把一个语法树结点对应的语法符号的属性称为该语法树结点的属性。

一个语法树结点的属性一般分为两类，一类是它的值是只根据这个结点的子节点的属性值计算得到的，这种属性称为“综合属性”；另一类是计算它的值的时候还需要用到父结点或者兄弟节点的属性值的，这样的属性称为“继承属性”。在自底向上的语法树遍历过程中一般只使用综合属性；在自顶向下的语法树遍历过程中通常需要使用继承属性。

关于属性文法的详细内容，我们将在后面的课程中进一步介绍，在 PA2 中我们只需要对属性文法有一个感性的认识即可。实际上大家会发现属性文法的实现是非常简单的。

PA2 中使用的主要属性域如下表所示：

结点种类	属性名称	含义	可能取值	备注
TypeLiteral	type	类型对象	type/*	被继承
Expr	type	结果类型	type/*	被继承
LValue	lvKind	左值种类	LValue.Kind	被继承
	type	结果类型	type/*	
TopLevel	main	Main 类符号	symbol/Class	
	gscope	全局作用域	scope/ GlobalScope	
ClassDef	symbol	类定义符号	symbol/Class	
MethodDef	symbol	函数定义符号	symbol/Function	
VarDef	symbol	变量声明符号	symbol/Variable	
CallExpr	symbol	函数符号	symbol/Function	
Ident	symbol	变量符号	symbol/Variable	
NewClass	symbol	类定义符号	symbol/Class	

这个表的备注栏中“被继承”的意思是说这些属性会被该结点的所有子类结点所继承，例如 Expr 的 type 属性将会被所有 Expr 类的接点继承，比如 Binary，CallExpr 等等。

LValue 的 lvKind 属性是指这个左值符号的种类，例如变量引用可能是局部变量引用，也有可能是成员变量引用，引用的具体分类参考 LValue 的一个内部 enum Kind。

设置属性值的时候请大家注意：保存属性值的目的是为了以后的计算使用的，因此我们并没有规定每种属性的值必须是什么，也就是说，大家可以设成最便于自己操作的值。

符号表组织

符号表的作用是管理符号信息，其主要目的是为符号信息的存储和访问提供一种高效、方便的途径。一个符号依据不同的场合可能有多种不同的属性，但是一般来说都具备两种基本属性：符号的名字、符号有效的作用域。

为了高效地从符号名字找出对应的符号实体，一般采用哈希表或者平衡二叉树（例如红黑树）又或者 trie 这样的数据结构来组织单个符号表。

符号的作用域信息用于帮助在同名的符号中找出当前有效的符号，也用于判断作用域错误。所谓的作用域是说一个符号在源程序中的有效范围。按照记录作用域信息的不同方法，符号表有两种组织方式：

1、单表形式

所有的标识符都记录在同一个符号表中，同时为每个记录添加一个作用域属性来标记该记录对应的标识符所声明的作用域层次。作用域的层次是用一个计数器来记录的，每进入一个新的作用域，计数器就增 1；每退出一个作用域，计数器就减 1。利用哈希表实现单表结构的符号表时，同名的标识符按照作用域层号递减的顺序记录在同一个桶对应的链表中。这样在访问一个标识符时，从哈希表中找到的第一个表项就是最靠近内层的作用域中声明的标识符。这种结构的最大开销在于每退出一层作用域时，都需要遍历整个符号表以删除该作用域中声明的所有标识符表项。

2、多级符号表

为每个作用域单独建立一个符号表，仅记录当前作用域中声明的标识符。同时建立一个栈来管理整个程序的作用域：每打开一个作用域，就把该作用域压入栈中；每关闭一个作用域，就从栈顶弹出该作用域。这样，这个作用域栈中就记录着当前所有打开的作用域的信息，栈顶元素就是当前最内层的作用域。查找一个变量时，按照自顶向下的顺序查找栈中各作用域的符号表，最先找到的就是最靠近内层的变量。这样处理的不利之处在于，查找外层变量的时间开销会变大。但这样组织的符号表实现起来比较简单，并且一般来说，作用域嵌套的层次不会太深，另外程序访问当前作用域中变量的概率总是大一些。

在 Decaf 项目中，我们采用第二种符号表组织方式。在实验框架里已经给出了符号表的完整实现，不需要任何修改就可以完成 PA2 的实验。不过我们还是建议你阅读一下这部分代码以了解该符号表的工作方式，并且你也自己添加一些辅助函数来更好的完成实验。

如果你不满足于框架程序中符号表的设计，可以采用自己设计的符号表组织方式，但是要保证输出与标准输出一致，同时工作效率不应有太大的损失。并请提供详细的设计文档，说明设计的理由，例如你的设计主要是为了满足什么样的需求，有何优点和不足等。

Visitor 模式

在 PA2 种我们将使用到 Visitor 这种设计模式。所谓设计模式是为了解决某一类问题而被反复使用的设计，用俗语来说就是针对一类问题的“套路”。我们这次使用的 Visitor 模式，主要用在以下场合：需要处理的对象种类相对固定，但需要对这些对象做很多不同的操作。而 Decaf 编译器恰恰就是这种情况——AST 结点的种类是根据语言规范确定的，基本不会有变动，但我们需要对 AST 结点做多趟遍历，以完成建立符号表，语义检查，转换为中间表示（PA3）等操作。下面我们针对 PA2 种需要处理的问题来简要的介绍一下 Visitor 模式的主要思想和处理方法。

在 PA2 中，我们首先需要对 AST 进行一次遍历，建立符号表，那么符合面向对象思想的处理方法，就是在 Tree 类中增加一个叫 buildSym 的 abstract 方法，所有的 AST 结点都重写这个方法，针对自己结点进行建立符号表的操作。我们在遍历 AST 结点的时候调用对每个结点调用 buildSym 方法完成建立符号表的操作。要注意 AST 结点非常多，因此这是一个相当费时且容易出错的过程，但是还可以忍受。那么，接下来，我们需要再对 AST 进行一次遍历，进行语义检查，按照上面的“面向对象”处理方法，Tree 中再增加一个 typeCheck 方法，每个 AST 结点重写这个方法，于是我们又要对几十个文件进行修改。接下来，PA3 中，我们又需要对 AST 结点进行遍历，完成到中间表示的转换，于是……

不停的修改一个抽象类或者接口并不是一种良好的编程习惯。那么，常见的想法就是把每趟遍历的方法抽出来写到一个类里面，比如对于语义检查的遍历过程，我们可以这样写一个类：

```
public class TypeCheck {
    public void typeCheck(Binary binaryExpr) {
        ……
    }
    ……
}
```

在遍历 AST 结点时，针对不同的结点调用不同的方法。

看上去似乎是可行的，但实际上这种处理方式有一个无法回避的问题，就是我们在遍历 AST 结点时，很多时候在编译期间是无法得知一个结点的确切类型的。考虑上述 typeCheck(Binary binaryExpr)的实现，要检查这个加法表达式是否合法，我们需要知道它两个操作数的类型，也就是针对其两个操作数调用 typeCheck 方法。问题来了，Binary 的两个操作数都是 Expr 类型的，而我们显然不会写一个针对 Expr 的 typeCheck 方法，因为 Expr 是一个抽象类，只有具体到一个 Binary 或者 Unary 等子类上时，我们才知道如何对其进行类型检查。有的同学可能想到了虚函数，但虚函数对于这种情况是无能为力的。对于 Java，C++这类编译器，在编译期间必须能明确绑定到需要调用的函数上（Decaf 也是这样，PA2 就有这部分内容），因此调用 typeCheck(Expr)时，编译器会去查找有没有 typeCheck(Expr)或者 typeCheck(Expr 的父类)这种方法，而我们显然不会写这种方法，于是……

一种比较龌龊但确实可行的方法，就是提供一个 typeCheck(Tree)的方法，在这个方法中，判断结点的类型，分别调用对应的方法。对于 Java 这种支持反射的语言，可以使用 instanceof 来判断结点类型，对于 C++，我们可以在 Tree 中增加一个表示结点种类的域用来判断结点类型。这种处理方法确实是可行的，但显然不是一种好的解决方法，因为用来分发请求的方法会异常臃肿，而且不符合面向对象的思想。

实际上，visitor 模式正是为解决这种问题而出现的一种设计模式。它使用了一种叫做双重分派（double-dispatch）的技术来解决这个问题。我们修改一下 Tree，为其增加一个 accept(TypeCheck typecheck)的 abstract 方法，所有子类都重写这个方法（不要怕，使用 visitor 模式只需要辛苦这一次），方法体都是一样的，只有一句话，typeCheck.typeCheck(this)。我们在遍历 AST 结点时，调用结点的 accept 方法，通过 accept 函数的一次指派，就能正确的调用 TypeCheck 中的对应方法了。

这是为什么呢？我们分析一下调用 accept 函数的过程，首先，accpet 函数是一个虚函数，在运行期能够正确的调用子类的相应方法，而在子类的方法体中，this 的类型是明确的，因此编译器能够明确的知道 typeCheck.typeCheck(this)应该调用哪个函数，这样问题就得到了圆满的解决。

现在我们给出 Visitor 模式的完整解决方案。首先声明一个 Visitor 接口：

```
public interface Visitor {
    // 为每一种结点声明一个方法
    public void visitBinary(Binary binaryExpr)
    ……
}
```

然后，在每个需要被处理的结点中定义一个 accept 方法（常作为 abstract 方法放到基类中），accept(Visitor visitor)，以声明自己可以接受一个访问（这就是

Visitor 模式为什么被叫做 Visitor 模式的原因），方法体如前所述，visitor.visit(this)。于是，BuildSym，TypeCheck 只要分别实现这个接口，就能够完成相应的遍历过程，而不需要对 AST 结点类做任何修改。之后的遍历过程也是如此。

实际上，框架中的 BuildSym 和 TypeCheck，并没有实现 Visitor 接口，而是继承自 Visitor 抽象类。使用接口的一个问题就是，实现接口时，需要将接口的所有方法都实现（不仅仅是特定编译器的限制，在逻辑上也应该是这样的）。但在很多情况下，一次处理过程可能并不需要处理所有类型的对象（考虑 BuildSym，它只需要处理很小的一部分 AST 结点），这样会导致很多空方法的出现。这显然不够优雅，并且写起来很麻烦（使用 IDE 的话一般没有这个问题）。提供一个所有方法都为空的父类可以避免这个问题，但需要注意不要漏掉本应该处理的 AST 结点。

提示

1、在 Decaf 中，类以及类成员的声明是无序的，也就是说所使用到的类或者类成员可以在后面的代码中才定义。这种特性使得程序员写程序的时候更加方便。如果把建立符号表和类型检查的过程合并在对 AST 的一趟扫描中完成的话要实现这样的特性非常困难（在本课程的后面我们将学习到一种叫做“拉链—回填”的办法，可以解决这个问题），而且这样的实现使得语义分析过程的逻辑非常混乱，很难保证其正确性。因此，我们采取了把这两项工作划分到不同的两趟 AST 扫描中，在第一趟扫描的时候建立好所有类以及其所有成员的符号信息，第二趟扫描的时候只需要进行简单检查即可。特别地，由于一个类的成员定义可能利用到一些还没有被定义的类，因此我们需要在为类成员建立符号信息之前事先把所有的类名扫描一次，并把合适的类定义信息记录在符号表中。

2、由于类的声明是无序的，因此有一种可能的问题是类 A 继承了类 B，而类 B 又继承了类 A；甚至是类 A 继承了类 A 自己。明显，这样的继承关系是不合法的，这种循环继承的错误应该在建立符号表的过程中在合适的地方进行检测。如果我们用图的一个节点表示一个类定义，用一条从结点 B 指向结点 A 的有向边表示类 A 继承于类 B 的话（这幅图就是所谓的“继承图”，Inheritance Graph），则一幅合法的继承图应当是一幅有向无环图（单继承情况下则成为一棵树）。检测一幅有向图是否无环的方法同学们应该在图论课中已经学习过，我们在代码框架中示例性地给出其中一种检测环路的方法，大家也可以自己实现其他的方法。

3、在 PA2 中，我们除了需要建立符号表以及进行类型检查以外，还需要对不合法的情况进行报错处理。在什么时候需要报错，报什么样的错误是一个比较不好处理的问题。有的同学可能会采取漫天联想来构造例子的做法，这种做法固然可以找出一些可能的情况，但是这种做法的效率是非常低的（而且还很容易导致思维混乱）。最正规的方法，应该是根据 Decaf 语言规范，总结出语义检查时需要遵循的规则，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。当然，我们在实验框架中已经给出了 PA2 中需要检测的所有的错误，并且也都给出了示例，因此，一种可能更快速的方法就是直接从需要报的错误入手，看每种错误违反了那种规则，然后找到和该规则相关的语句，进行报错。

请注意，报错的目的是为了给程序员提出更正错误的提示，因此我们应当把关键的错误用法都报出来。但是另一方面，有些错误是因为前面的错误引起的，如果把所有的可能错误都报出来的话程序员将会因为垃圾信息过多而没法分出哪些才是关键的错误。因此实际上各种编译器的实现都是只报关键的错误，而把可能由前面错误引起的那些后续错

误省略掉。报错的完整性和扼要程度是需要取舍的，关于这个取舍问题请参考后面关于连锁报错的提示。

4、在代码框架中，我们故意在语义分析的两个阶段中间插入了一个检查点：在这个检查点之前如果曾经报错，则编译过程会立即终止，仅当到达这个检查点的时候仍然没有发现任何错误才继续执行。这是一种简化措施，目的是保证开始类型检查的时候符号表的信息都是正确而且可靠的。如果不加入这样的检查点的话，在建立符号表的过程中，一旦遇到非法用法，除了需要报错以外还需要提供一定的修复错误或者标记手段，这给第一趟和第二趟扫描的实现都带来了一定的复杂度。

5、避免连锁报错：有时源程序的一个语义错误可能引起连锁反应，导致其它地方也出现语义错误。这种情况下如果在每个地方都报错，可能会引起程序员的误解，所以我们建议大家只在错误的根源处报错，对由此引起的其它错误不再报错。例如，对于一个未声明的标识符 b，对 b[4]应当怎样处理呢？对 b[4]+5 又应当怎样处理呢？如果我们认为 b 是一个整型数组类型的变量，只是漏了声明语句，那么 b[4]和 b[4]+5 就不必再报错；又比如，声明了一个函数原型 func(int a, int a)，形参表中有同名参数，显然有错。可是如果后面出现 func(2,3)是否要报错呢？如果程序员把形参表参数重名的错误改正的话，func(2,3)是完全合法的，那就没必要对 func(2, 3)报错了。这里向大家介绍一个小技巧，就是可以用 BaseType.ERROR 来表示因为出错而导致无法推断的数据类型，一旦遇到 BaseType.ERROR 就说明前面已经报过错，不必再报了。另外需要注意的是，当表达式中一个操作数的数据类型为 BaseType.ERROR 时，如果这个表达式的运算结果类型只可能有一种(例如&&,>这种操作,返回值只可能是bool)，那么我们就不需要把 BaseType.ERROR 传递开去（否则后面一些潜在的错误就会被忽略了），因为我们知道即使程序员把前面的错误更正了，这条表达式的运算结果类型也只可能是所规定的那种。