第五讲　实验导引（二）

2021-10

1. MiniDecaf 实验框架简述

1.1　回顾：实验框架分五个阶段

我们将 MiniDecaf 项目的实验框架分成如图 1 所示的 5 个阶段：

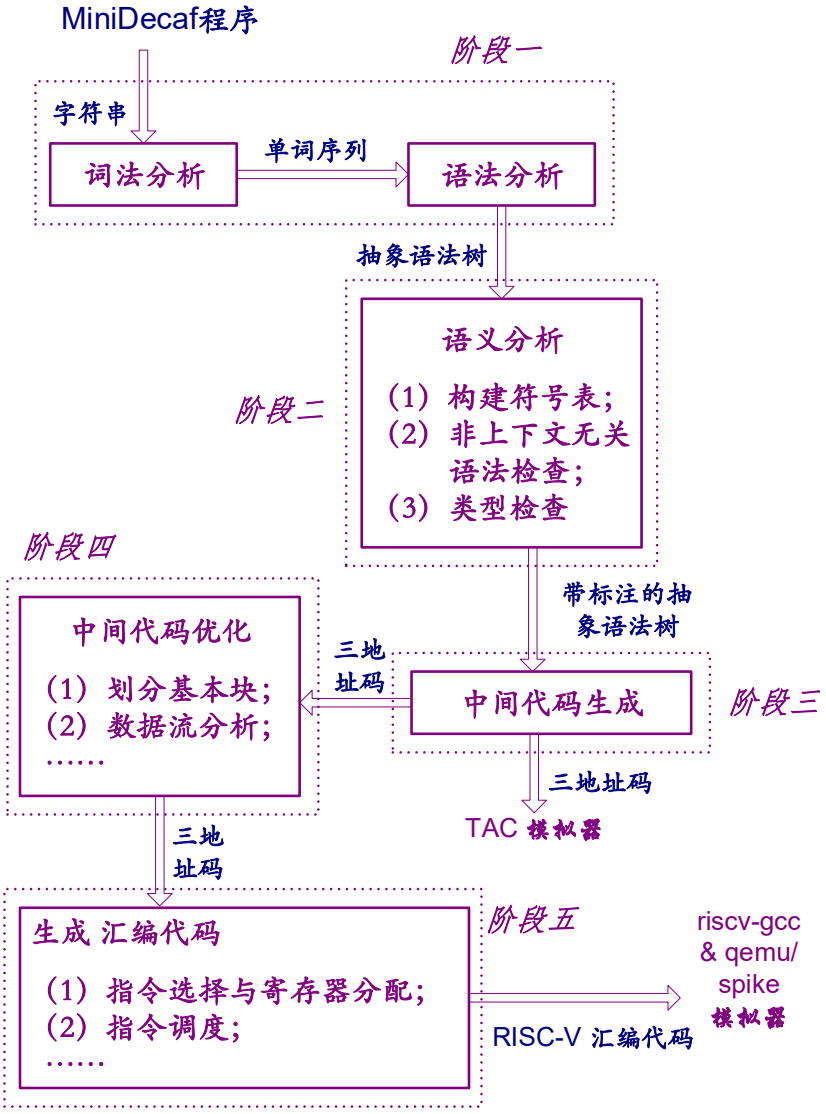

图 1　MiniDecaf 项目实验框架的五个阶段

如图 1 所示，阶段一进行词法和语法分析，并产生一种高级中间表示（实验指定的抽象语法树 AST）；阶段二为语义分析；阶段三生成三地址码；阶段四实现一些简单的数据流分

析；阶段五生成汇编代码。

1.2 阶段一（PA1）：词法/语法分析

在阶段一中，我们将 MiniDecaf 程序转化为抽象语法树。在词法分析中，我们从左到右扫描 MiniDecaf 源程序，识别出标识符、保留字、整数常量、算符、分界符等单词符号（即终结符），并返回给语法分析器使用。在语法分析中，我们针对所输入的终结符串建立抽象语法树，并对不符合语法规则的 MiniDecaf 程序进行报错处理。词法分析、语法分析和抽象语法树相关源代码的位置如表 1 所示。

表 1　词法/语法分析源代码在框架中的位置

源代码位置	词法分析	语法分析	抽象语法树
C++框架	用于生成词法分析程序的 Lex 代码位于 src/frontend/scanner.l	用于生成语法分析程序的 Yacc 代码位于 src/frontend/parser.y	src/ast/文件夹
Python 框架	语法分析程序位于 frontend/parser	语法树相关代码位于 frontend/ast	ast

1.3 阶段二（PA2）：构造符号表、实现静态语义分析

语法正确只说明了所输入源程序在格式上是合法的，但是要进行有效的翻译，编译器还需要了解输入中所出现的每个标识符所代表的内容，同时需要明确输入程序各部分的含义，这一过程称为语义分析。

在阶段二中，我们将遍历前一阶段产生的抽象语法树，进行符号表构造和类型检查，然后产生带标注的抽象语法树（decorated AST）。这一阶段将对抽象语法树 AST 进行两趟扫描：第一趟扫描的时候构建符号表（对应于 C++框架的 SemPass1 类和 Python 框架的 Namer 类），为后续的代码生成收集及建立必要的类型信息，相关信息存入符号表或 AST 结点（符号表也存在对应的 AST 结点上，比如 C++框架中，CompStmt 结点和 ForStmt 结点上都保存了符号表信息，而 Python 框架中，Block 结点和 For 结点上也保存了符号表信息），并且进行语义规范检查，比如检测符号声明冲突等问题（例如同名变量在同一作用域中不能多次定义，不能引用未定义变量等）；第二趟扫描的时候检查所有的语句和表达式中参数的数据类型。在进行语义分析的过程中，需要进行**静态语义检查**，即审查程序是否符合源语言所规定的语义规则（注意，在程序执行过程中所进行的语义检查称为**动态语义检查**），发现不符合语义规则时报告语义错误。例如，图 2 中的 MiniDecaf 程序片段是符合语法规则的，但不符合语义规则。

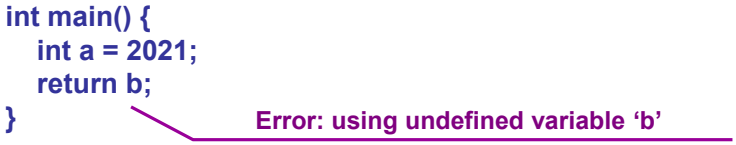

图 2　语义检查

此外，还有如下静态语义分析的例子：

（1）运算数与给定运算不兼容（一般在第二趟扫描中进行）

int a[3]; a * 2;

（2）函数调用实参数量与函数的形参数量不匹配（一般在第二趟扫描中进行）

int fun() { return 1; }

int main() { return fun(1); }

（3）左值检查（可以在第一趟扫描中，也可以在第二趟扫描中）

(a = 1) = 1 // a=1 不是左值，所以出错

（4）检查是否有 main 函数（可以在第一趟扫描或第二趟扫描中进行）

（5）检查 break/continue 是否在 for、while 等循环结构中（可以在第一趟扫描或第二趟扫描中进行）

实现静态语义检查时，我们需要一个重要的数据结构，即**符号表**。符号表可以体现符号的含义以及作用域等信息。随着语义分析过程的进行，当遇到符号定义的时候，我们需要向符号表中加入适当的记录信息；当遇到对符号的引用时，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含义的符号。如果引用了一个没有定义的符号，我们将无法得知这个引用的具体含义，后续的翻译阶段也无法正常工作。因此，需要把不合法的符号引用筛选出来。同样，如果有两个相同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无法得知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。诸如此类的语义问题，看似语法问题，但其正确形式是无法用上下文无关文法来表达的，因此，我们称之为**非上下文无关的语法检查**。

对于语句或者表达式的含义，我们需要根据 MiniDecaf 语言规范中的语义规定来检查其是否合法。一般情况下，编译器对于语句和表达式只进行类型检查，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。为简化实验框架，MiniDecaf 中标识符的类型仅有三种：int 类型（32 位整数）、int 数组类型（支持任意多维的整数数组，数组的长度必须是一个正整数字面量）、函数类型。为支持 int 数组，在 Python 框架中定义了 ArrayType 类（frontend/type/array.py），在 C++框架中定义了 ArrayType 类（type/type.hpp 内）。仅支持声明 int 和 int 数组两种类型的变量，不支持函数变量。对函数类型而言，为简化实现，仅支持参数类型及返回值均为 int 类型的函数。在 Python 框架中，并未被显式定义对应于函数类型的类，在 C++框架中定义了 FuncType 类（type/type.hpp 内），不同框架的实现略有区别。

语义分析过程中所收集的有用信息将记录在抽象语法树的相应结点中，对于语义正确的程序来说，就得到一棵带标注的抽象语法树。带标注的抽象语法树及其关联的符号表信息将传递到后续阶段使用。

例如，对于以下 MiniDecaf 程序片段：

```
int a = 0;
int func(int a) {
    int b = 1;
    return a + b;
```

```
}
int main() {
    int a = 2021;
    if (a) {
        int a = 0;
        int b;
    }
    return a;
}
```

其相应的抽象语法树（或带标注的抽象语法树，差别只是后者附加的信息更加丰富）形如（此处以缩进表示 AST 结点的所在的层，program 结点为 AST 的根节点）：

```
program
  declaration
    type(int)
    identifier(a)
    int(0)
  function
    type(int)
    identifier(func)
    param_list
      parameter
        type(int)
        identifier(a)
    block
      declaration
        type(int)
        identifier(b)
        int(1)
      return
        binary(+)
          identifier(a)
          identifier(b)
  function
    type(int)
    identifier(main)
    param_list
    block
      declaration
        type(int)
        identifier(a)
        int(2021)
      if
        identifier(a)
        block
```

```
        declaration
          type(int)
          identifier(a)
          int(0)
        declaration
          type(int)
          identifier(b)
          NULL （未定义的初值）
      NULL （对应if的else分支）
    return
      identifier(a)
```

其相应的符号表与作用域信息可能被描述为：

名称	类别
a	variable
func	function
main	function

全局作用域

名称	类别
a	variable
b	variable

func 作用域

名称	类别
a	variable

main 作用域

名称	类别
a	variable
b	variable

if 分支作用域

实验中，符号表采用多级结构，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。有两种类型的作用域，即全局（Global）作用域和局部（Local）作用域。

当处理到程序的某一位置时，可以访问的作用域称为开作用域，否则为闭作用域。需要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，按照自栈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，恰好是想要引用的变量。

如图 3 所示，当处理到 int b 语句时，当前作用域栈中包含 3 个开作用域。

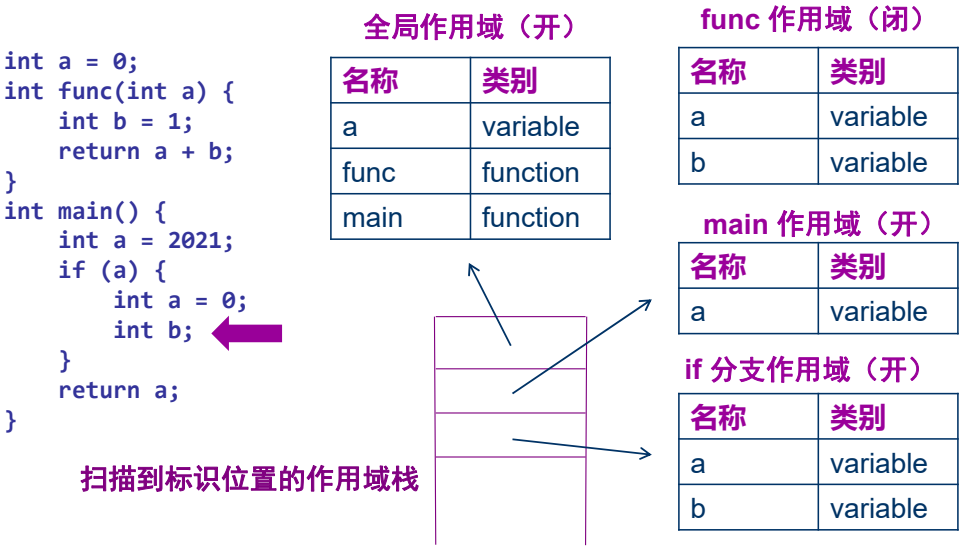

图 3　作用域栈

符号表构建相关源代码的位置如表 2 所示。类型检查相关源代码的位置如表 3 所示。

表 2　符号表构建源代码在框架中的位置

源代码位置	符号表构建	符号表数据结构	作用域数据结构
C++框架	符号表构建位于 src/translation/build_sym.cpp	符号表相关的数据结构位于 src/symb	作用域相关数据结构位于 src/scope
Python 框架	符号表构建位于 frontend/typecheck/namer.py	符号表相关的数据结构位于 frontend/symbol	作用域相关数据结构位于 frontend/scope

表 3　类型检查在框架中的位置

源代码位置	类型定义	类型检查
C++框架	位于 type/文件夹中	位于 translation/type_check.cpp
Python 框架	位于 frontend/type/文件夹中	位于 frontend/typecheck/typer.py

1.4 阶段三（PA3）：中间代码生成

在阶段三中，我们将带有标注的抽象语法树翻译成适合后端处理的中间表示。MiniDecaf 实验使用的中间表示是**三地址码**（Three-Address Code，TAC）：这是一种类似于汇编的中间表示，所有指令至多包含三个操作数。三地址码和汇编语言最大的差别在于汇编中使用的是目标平台 RISC-V 规定的物理寄存器，其数目有限；而三地址码使用的是“伪寄存器”（或**临时变量**），数目不受限制。在后端生成汇编代码时，我们再考虑如何为临时变量分配物理寄存器的问题。

下面给出了一个 TAC 程序的示例：

```
main:                   # main 函数入口标签
    _T0 = 1             # 加载立即数
    _T1 = _T0           # 临时变量赋值操作
    _T2 = ADD _T0, _T1  # 加法操作 _T2 = _T0 + _T1
    _T3 = NEG _T0       # 取负操作 _T3 = -_T0
    return _T3          # 函数返回
```

TAC 程序由标签和指令构成：

（1）标签用来标记一段指令序列的起始位置。从底层实现的角度来看，每个标签本质上就是一个地址，且往往是某一段连续内存的起始地址。在我们的实验框架中，标签有两个作用：作为函数入口地址（如上例中的 main 函数入口），以及作为分支语句的跳转目标（TAC 指令不支持 MiniDecaf 语言中条件和循环控制语句，而是将它们都翻译成更加底层的跳转语句）。

（2）TAC 指令与汇编指令类似，每条 TAC 指令由操作码和操作数（最多 3 个）构成。操作数包括：临时变量、常量、标签（可理解为常量地址）和全局变量。如上例所示，TAC 中的临时变量均用 "_Tk" 的形式表示（k 表示变量的编号）。

下面给出 Python 框架中生成 While 循环 TAC 指令的源程序（其中的 accept 函数调用采用了接下来要讲解的 Visitor 设计模式）：

```
def visitWhile(self, stmt: While, mv: FuncVisitor) -> None:
    …
    # 在 FuncVisitor 中开启一个循环（用于 break/continue 的代码生成）
    mv.openLoop(breakLabel, loopLabel)
    mv.visitLabel(beginLabel) # 生成循环体开始的跳转标签
```

```
    stmt.cond.accept(self, mv) # 生成 while 条件判断的代码
    mv.visitCondBranch(tacop.CondBranchOp.BEQ, stmt.cond.getattr(“val”),
breakLabel) # 根据条件判断结果添加条件跳转指令
    stmt.body.accept(self, mv) # 生成 while 循环体的代码
    mv.visitLabel(loopLabel) # 生成循环体执行的跳转标签（用于 continue）
    mv.visitBranch(beginLabel) # 跳转到循环体开始的标签
    mv.visitLabel(breakLabel) # 生成退出循环体的标签
    mv.closeLoop() # 结束当前循环
```

所生成的 While 循环 TAC 程序如下：

```
    begin:
        <cond>
        if <cond>.val == 0: goto break
        <body>
    loop:
        goto begin
    break:
```

同样的 While 循环，其 C++框架中生成 TAC 指令的源程序（其中的 accept 函数调用也采用了接下来要讲解的 Visitor 设计模式）：

```
    void Translation::visit(ast::WhileStmt *s) {
        Label L1 = tr->getNewLabel(); // 新建 while 循环执行和退出的标签
        Label L2 = tr->getNewLabel();
        Label old_break = current_break_label;
        current_break_label = L2; // 进入当前循环，修改 break 的跳转位置
        tr->genMarkLabel(L1); // 生成循环体执行的标签
        s->condition->accept(this); // 生成 while 循环条件的代码
        // 根据条件判断结果添加条件跳转指令
        tr->genJumpOnZero(L2, s->condition->ATTR(val));
        s->loop_body->accept(this); // 生成 while 循环体的代码
        tr->genJump(L1); // 跳转到循环体开始的标签
        tr->genMarkLabel(L2); // 生成退出循环体的标签
        current_break_label = old_break; //结束当前循环，break 的跳转位置复原
    }
```

所生成的 While 循环 TAC 程序如下：

```
    L1:
        <condition>
        if <condition>.val == 0: goto L2
        <loop_body>
        goto L1
    L2:
```

由两个框架所生成的 TAC 程序可见，同样是 While 循环，可以设计出不同的 TAC 代码。同时，框架中提供的源代码是可以处理嵌套 While 循环的，相关内容，我们在后续课程中会进一步介绍。

TAC 程序是无类型的，或者说它仅支持一种类型：32 位（4 字节）整数。为了简化实验内容，MiniDecaf 中标识符的类型仅有三种(int 类型、int 数组类型和函数类型)，仅能声明 int 和 int 数组两种类型的变量,不支持函数变量。我们规定 MiniDecaf 程序中的 int 类型的变/常量与 TAC 中的变/常量可以直接对应，简化了中间代码乃至目标代码的生成过程。数组类型对应于一段连续的 32 位整数单元, 虽无法用单个临时变量直接表示，但可以用一个临时变量作为起始地址，并借助另一个临时变量或立即数作为偏移量（对应于数组下标*4 字节）实现对数组元素的访问，或将其作为左值进行赋值。对于函数类型，其每个调用实例都将对应一个由控制单元（保存函数调用与返回相关的信息）和数据单元构成的栈帧空间，每个控制单元与/或数据单元均可看作32位整数单元,对其访问可通过基地址加偏移量的方式实现。TAC 生成相关源代码的位置如表 4 所示。

表 4　TAC 生成在框架中的位置

源代码位置	三地址码定义	三地址码生成
C++框架	位于 src/tac 文件夹中	位于 src/translation/translation.cpp
Python 框架	位于 src/tac 文件夹中	位于 frontend/tacgen/tacgen.py

1.5 阶段四（PA4）：中间代码优化

阶段四进行中间代码分析与优化（为了简化实验框架，以分析为主），其分析结果对后续的寄存器分配阶段具有重要的支撑作用。在实验框架中，该阶段主要进行控制流分析和数据流分析，求得**活跃变量**信息。要进行数据流分析，首先要进行基本块划分（目标代码生成也是以基本块为单位进行的），然后进行控制流图构造，接下来在控制流图上可以建立活跃变量数据流方程，并完成活跃变量分析。活跃变量分析的基本思想是判断某个变量在程序某点是否还活跃，即在该点之后是否还会被引用。有了活跃变量信息，就可以对每个变量所用的寄存器进行分配和调度，不再活跃的变量，可以收回其所占用的寄存器，从而提高寄存器的使用效率。这部分内容在框架中已有提供，有兴趣的同学可以自行阅读参考，在后续讲解数据流分析相关内容时，将会有更深入的理解。数据流分析相关源代码的位置如表 5 所示。

表 5　数据流分析在框架中的位置

源代码位置	数据流分析
C++框架	位于 src/tac/flow_graph.cpp 及 src/tac/dataflow.cpp 中
Python 框架	位于 backend/dataflow/ 文件夹中

1.6 阶段五（PA5）：目标代码生成

最后，我们将三地址码翻译成目标平台 RISC-V 上的汇编代码。主要包括两个步骤：寄存器分配和指令选择。

所谓寄存器分配，是指为中间代码中的虚拟寄存器分配实际的物理寄存器。对中间代码来说，通常假设虚拟寄存器的数量是无限的，这导致我们在分配物理寄存器时无法简单的对虚拟寄存器做一一映射，需要有一个调度与分配算法来合理使用有限的物理寄存器。前一阶段所得的活跃变量信息可以作为寄存器分配算法的输入，如果某个变量不再活跃（未来不再被引用），那么该变量所占用的物理寄存器即可收回。由此可见，活跃变量信息对于寄存器分配至关重要。

实验框架中已经实现了控制流和数据流分析，以及一种启发式寄存器分配算法供大家使用，这部分内容在课程实验中不做具体要求，感兴趣的同学可以自行阅读和学习。

在分配寄存器之后，我们需要将 TAC 指令翻译成对应的 RISC-V 汇编指令。此外，对于函数（过程）调用，需要进行栈帧管理。进入所调用函数时，需要开辟栈帧（通过 TAC 代码确定所需开辟的栈帧大小）；在从所调用函数返回时，需要恢复栈帧。其中涉及参数传递、函数返回地址、局部变量等问题，在后续讲授运行时存储组织时将详细介绍。目标代码生成相关源代码的位置如表 6 所示。

表 6　目标代码生成在框架中的位置

源代码位置	寄存器分配	目标代码生成	新增汇编指令
C++框架	位于 src/asm/riscv_md.cpp 中	位于 src/asm/riscv_md.cpp	修改 src/asm/riscv_md.cpp
Python 框架	位于 backend/reg/bruteregalloc.py 中	位于 backend/riscv/riscvasmemitter.py 和 utils/riscv.py 中	修改 utils/riscv.py

2. Visitor 设计模式简介

实验框架的第一阶段可以将 MiniDecaf 源码变换到相应的抽象语法树（AST）。实验框架的第二阶段和第三阶段中，将对 AST 进行多遍扫描，每一遍扫描都要针对 AST 的所有结点进行处理，处理到不同结点时将有不同的行为。为了便于实现这一需求，实验框架中采用了 Visitor 设计模式。

设计模式 [1] 是人们为解决同类设计问题总结出来的行之有效的软件设计定式。合理使用设计模式，可以使软件更加易于理解和维护。近年来，Visitor 模式在采用面向对象技术实现的编译系统中被广泛采用。下面我们结合本课程主体实验框架，简要介绍 Visitor 模式的设计思想。

Visitor 设计模式用于表示一个作用于某对象结构中各元素的操作。使我们可以在不改变各元素的类的前提下，定义作用于这些元素的新操作。图 4 给出 Visitor 设计模式的类图，包括两个基类 Element 和 Visitor。其中，ObjectStructure 类中包含多个元素（Element），每个具体元素（ConcreteElementA 等）都实现了 Accept 函数接口。ConcreteElementA 类的 Accept 函数定义为：v.VisitCrtElementA(this)；同理，ConcreteElementB 类的 Accept 函数定义为：v.VisitCrtElementB(this)。从图中可见，采用 Visitor 设计模式，可以很方便的加入多个 ConcreteVisitor，而每个 ConcreteVisitor 都可以对 ObjectStructure 中的所有元素完成访问。

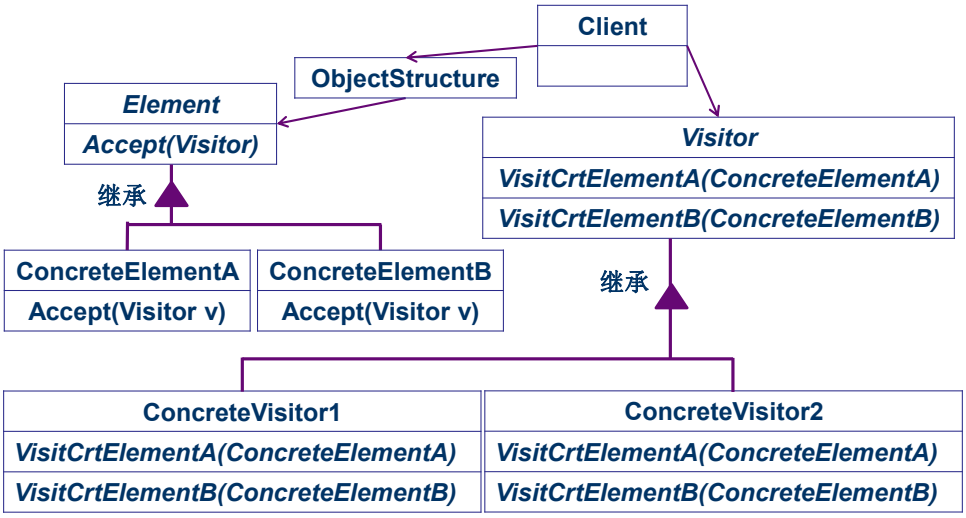

图 4　Visitor 设计模式类图

由于课程实验给大家提供了 C++和 Python 两种语言的框架，两种框架的实现大同小异，但是类的命名有些区别，为了便于讲解，我们在接下来的讲解中参考 C++框架中的类名，并进行一定的简化。

在课程实验框架中，AST 结点对应 Visitor 设计模式中的 Element 类，AST 结点构成的树型结构对应 Visitor 模式中的 ObjectStructure 类。AST 结点的种类有几十个，它们所对应的 class 都是类 ASTNode 的子类，而 ASTNode 中定义了虚函数 accept，如图 5 所示。树的根结点为 Program 对象，对 Program 对象的 visit 过程会遍历 Program 的子结点，从而完成对整个 AST 的遍历。实际上，AST 结点可以有多个抽象层次，对应类继承的多个层次。比如，Statement 类对应的结点，可以细分为 ReturnStmt、IfStmt、WhileStmt 等。但本节所讨论的内容与这些中间层次的抽象类关系不大，所以我们假设类层次结构只有如图 5 所示的两层，各种 AST 结点的 class 分别为 A，B，C，…等。

在阶段二中，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，就是在类 ASTNode 中增加一个叫 buildSym 的虚函数，所有的 class A，B，C，…，等都重写这个方法，针对自己结点的类别进行建立符号表的操作。我们在遍历 AST 结点的时候对每个结点调用 buildSym 方法完成建立符号表的操作。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，我们又需要对 AST 进行另一次遍历，进行语义检查，按照这种处理方法，ASTNode 中须再增加一个 typeCheck 方法，每个 AST 结点重写这个方法，于是我们又要对 class A，B，C，… 等进行修改。接下来，阶段三中，我们又需要对 AST 结点进行遍历，完成到中间表示的转换。可见，采用上面所述的设计方式，我们需要不停的修改每一个类，这不符合软件设计的开闭原则（open for extension，closed for modification），易于引入各种编程问题。

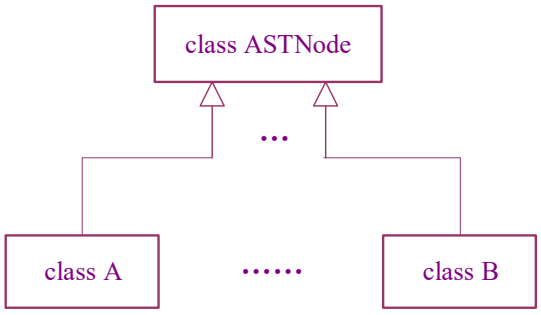

图 5　AST 结点类都是抽象类 Tree 的子类

采用 Visitor 模式可以有效地解决上述软件设计问题。这种方法将每一次对 AST 的遍历工作收集到一个单独的类（图 4 中的 ConcreteVisitor），而不是将这些工作分散至不同的 AST 结点。比如，我们把建立符号表的功能收集到一个类 BuildSym 中，然后针对每个 AST 结点 A，B，…，建立符号表的方法分别为 visitA，visitB，…，如图 6 所示。由于 C++语言支持重载，因此在参数不同的前提下，函数的名称可以相同，所以 visitA，visitB 等函数的名字可以修改为 visit。

```
class BuildSym
virtual void visitA (A *it) {
    ......
}
virtual void visitB (B *it) {
    ......
}
......
```

图 6　建立符号表的类 BuildSym

Visitor 模式提供一种所谓双重分派（double dispatch）的技术，可以简洁地支持这种将每一次遍历工作收集到一个单独 class 的解决方案，方法为：

1） 针对每个结点类 A，B，…，Visitor 类都对应有虚函数 visitA，visitB，…，如图 7 所示。

2） 每一次遍历工作对应的 class 都继承自类 Visitor，如图 8 所示。这些功能类将对方法 visitA，visitB，…，等进行覆写（override）。如图 6 所示，在类 BuildSym 中的 visitA，visitB，…，等将具体定义针对各个结点类（A，B，…，）实现建立符号表的功能。

3） 为抽象类 ASTNode 增加一个接受 Visitor 对象的方法，`void accept(Visitor *v)`，如图 9 所示。

每个结点类 A，B，…，等都覆写（override）方法 `void accept(Visitor *v)`，但对应的函数体十分相似，都只有一句话，对于类 A 是 `v.visitA(this)`，而对于类 B 是 `v.visitB(this)`，如图 10 所示。

```
class Visitor

virtual void visitA (A *it) {
    visit(it);
}
virtual void visitB (B *it) {
    visit(it);
}
……

virtual void visitProgram (ASTNode *it) {
    ……;
}
```

图 7 基类 Visitor

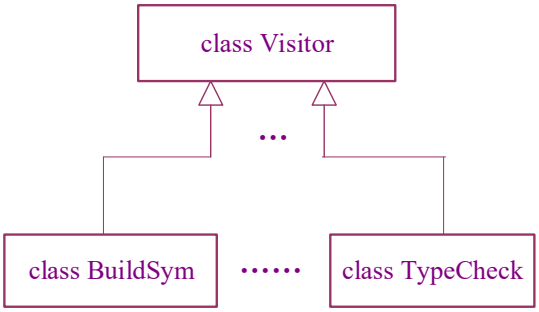

图 8　各次遍历工作对应的类都继承 Visitor 类

```
class ASTNode

virtual void accept(Visitor *v) {
    v->visitTree(this);
}
……
```

图 9　抽象类 ASTNode 中含一个接受 Visitor 对象的方法

```
class A

virtual void accept(Visitor *v) {
    v->visitA(this);
}
……
```

……

```
class B

virtual void accept(Visitor *v) {
    v->visitB(this);
}
……
```

图 10　每个结点类都重载接受 Visitor 对象的方法

这种设计的好处是，当我们需要新增加一遍扫描工作时，仅需增加一个新的类，通过继承 Visitor 类和覆写其中的方法，将这一遍扫描的工作收集到这个类中，而不影响代码的其

他部分。例如，如果我们需要对 AST 进行另一次遍历，进行语义检查，只需要增加一个新的类 TypeChecker，将语义检查的全部功能封装在其中，如图 8 和图 11 所示。

```
class TypeChecker

virtual void visitA (A *it) {
    ......
}
virtual void visitB (B *it) {
    ......
}
......
```

图 11　完成类型检查功能的类 TypeChecker

在课程实验的实际实现中，C++框架包括符号表构建（SemPass1 类）、类型检查（SemPass2 类）和三地址码生成（Translation 类）三遍扫描，在 Python 框架中包括符号表构建（Namer 类）、类型检查（Typer 类）、和三地址码生成（TACGen 类）三遍扫描，都采用了 Visitor 设计模式。

本节 Visitor 模式的代码结构是基于标准的 Visitor 模式来介绍的。其实，我们可以简化 Visitor 类中虚函数接口的定义，如[2]和[3]中介绍的 Visitor 类，只含一个通用的 visit 虚函数。

最后提一句，本节所介绍的设计模式最好应用于被访问的结点类别数目基本固定或是变化不太大的情形。在我们的实验框架中，含有几十个 AST 结点，近年来变化不大。

参考文献

1. Erich Gamma, etc., Design Patterns: Elements of Reusable Object-Oriented Software. Addison - Wesley, Reading, MA, 1995.
2. Charles N. Fischer, Ronald K.Cytron,　Richard J. LeBlanc, Jr., Crafting a Compiler, 清华大学出版社影印，2010。
3. Andrew W.Appel，Modern Compiler Implementation in Java ，人民邮电出版社影印，2005

课后作业

1　对 Visitor 设计模式相关内容进行调研，试整理一份较为全面的综述性文档。

（非书面作业，可选）