函数式编程语言编译

华保健
中国科学技术大学苏州高等研究院
中国科学技术大学软件学院
bjhua@ustc.edu.cn

目录

I 引言 1

II 闭包 2
- II-A 高阶嵌套函数的编译挑战 2
- II-B 环境与闭包 2
- II-C 闭包变换算法 3
- II-D 平坦闭包与链式闭包 5
- II-E 闭包与静态链 6

III 延续 6
- III-A 延续的基本概念 6
- III-B 延续传递风格 7
- III-C 延续传递风格变换 8

IV λ 演算的编译 9
- IV-A λ 演算 9
- IV-B 延续传递风格变换 10
- IV-C 闭包变换 11
- IV-D 函数提升 12
- IV-E 显式分配 12
- IV-F RISC-V 代码生成 13

V 优化 14
- V-A 内联展开 15
- V-B 尾递归优化 17

VI 静态单赋值形式与函数式编程 18
- VI-A 基本块与延续 18
- VI-B 静态单赋值形式转换为延续传递风格 19
- VI-C 延续传递风格转换为静态单赋值形式 20

VII 习题 20

参考文献 21

I. 引言

函数式编程语言是编程语言中历史最悠久、成果最广泛、应用最广泛的编程范式之一。函数式编程语言最早起源于对可计算理论的研究，在上世纪三十年代，Alonzo Church 提出了 λ 演算系统，奠定了最早的函数式编程的理论框架。在电子计算机出现之后，以早期的 LISP 为代表的不同风格的函数式编程语言被设计出来，并在各个领域得到了广泛应用。而面向大数据、云计算、函数即服务等新型应用场景和范式，函数式程序语言由于无状态和满足等式推理等先进特性，其重要地位日益提升。

相比较命令式或面向对象编程，函数式编程语言提供了更高层和抽象的编程机制，这使得它能更简洁的表达复杂的程序和算法。但函数式的这些编程特性，也给其编译器实现带来了新的问题和挑战。对函数式编程语言编译技术的研究，一直是研究的重点和热点，产生了丰富的研究成果。

深刻理解和掌握函数式编程语言的实现机制和技术，有重要的意义和价值。首先，理解和掌握函数式语言编译实现机制，有助于加深对函数式编程语言的把握和理解，从而能更好的利用这些语言进行程序设计实践。其次，也更重要的是，由于现代的主流编程语言都已经或正在走向多范式模式，函数式编程机制和命令式或面向对象语言不断融合，因此，理解和掌握函数式语言编译实现机制，也对更好的理解和掌握目前以及未来的多范式编程语言具有积极的促进作用。

本文深入讨论函数式语言的编译技术。首先，本文讨论了针对高阶嵌套函数的闭包编译技术，对闭包及环

境、闭包变换、平坦闭包与链式闭包、以及闭包与静态链等都做了深入讨论。接下来，本文讨论延续，对延续的基本概念、延续传递风格、以及延续传递风格的变换做了深入分析。接着，本文结合 λ 演算，并以 RISC-V 为目标指令集体系结构，讨论了一个小的函数式编程语言编译器的完整实例。接下来，本文对静态单赋值形式和函数式编程的内蕴联系进行了深入讨论，建立了静态单赋值形式和延续传递风格的对应关系。最后，本文还提供了丰富的习题以及参考文献，方便读者进一步理解和学习。

II. 闭包

闭包是函数式编程语言中一个非常重要的概念，也是对高阶嵌套函数语言特性的重要实现技术。闭包是指一个函数可以访问并操作其词法作用域之外的值或变量：当函数被定义时，闭包可以动态捕获其外部执行环境的执行状态；而在该函数被调用时，闭包中的值可以在函数调用过程中使用。闭包使得函数可以在不同的上下文中使用，而不需要将这些上下文状态作为显式参数传递进来，从而令函数更灵活且有更强的表达能力。本节我们将讨论闭包的主要概念，主要内容包括高阶嵌套函数的编译挑战、环境与闭包、闭包变换、以及链式与平坦闭包等。

A. 高阶嵌套函数的编译挑战

在支持函数式编程特性的编程语言中，函数经常被用作一阶对象，即函数可以赋值给其它变量，可以被当做参数传递，也可以作为返回值，等等；正因为这个原因，函数在这类语言中经常被视为一等公民。进一步，如果函数还可以被嵌套定义，则这类函数称为*高阶嵌套函数*。

考虑图 1中给出的高阶嵌套函数示例（为简化起见，我们省略了函数的返回值类型等细节信息，它们不影响本文的技术讨论）。其中函数 `a` 接受一个参数 `x`，并返回一个函数 `b`，因此函数 `b` 是一个一阶对象；同时，函数 `b` 嵌套定义在函数 `a` 内部，因此函数 `b` 是一个高阶嵌套函数。类似的，函数 `c` 也是一个高阶嵌套函数。

```
1  a(int x){
2    b(int y){
3      c(int z){
4        return x + y + z; }
5      return c; }
6    return b;
7  }
8  f = a(3);
9  g = f(4);
10 h = g(5);
```

图 1: 高阶嵌套函数示例

高阶嵌套函数对变量的使用模式更加复杂。高阶嵌套函数不但可以访问自身定义的变量，还可以访问其外层函数中的变量。例如，函数 `c` 不但用到了其自身的参数变量 `z`（第 4 行），同时也用到了变量 `x` 和 `y`（第 4 行），但是变量 `x` 和 `y` 并不在函数 `c` 的词法作用域中，而是分别位于其外部的函数 `a` 和 `b` 中（第 1、2 行）。如果一个变量 x 在一个函数内部使用，但未在该函数内部定义，则称该变量 x 为*自由变量*（Free variable）。自由变量的值不是由函数定义中的参数或局部变量确定，而是由函数调用时的执行上下文确定。

高阶嵌套函数给编译器实现带来的新的挑战。编译器在编译普通函数时（类似 C 语言中的函数），通常将其直接编译为一个内存地址，该地址中存放有该函数编译得到的代码；并且，该地址可以被当做参数传递、进行赋值、或者被当做函数的返回值。但是，由于自由变量的存在，这种将函数编译为内存代码地址的简单编译策略，无法直接用于高阶嵌套函数的编译。考虑上述代码示例中的函数调用 `f=a(3)`（第 8 行），如果函数直接被编译为其内存地址，则当该函数调用返回后，变量 `f` 等于函数 `b` 的地址；接下来，程序执行函数调用 `g=f(4)`（第 9 行），则开始执行函数 `b`，而函数 `b` 需要访问其自由变量 `x`，但是，由于函数 `a` 的调用已经完成，其参数 `x` 随即消失，因此函数 `b` 无法访问得到变量 `x`，从而导致执行出错。

B. 环境与闭包

环境（Environment）是指一个函数所有自由变量的列表。在函数体内，需要加入环境作为其额外的参数，该参数指明函数体如何访问其自由变量；而当函数被定义时，则需要创建该执行时刻的函数执行环境，并和函数指针绑定在一起。

闭包（Closure）是将函数的环境和函数指针组合在一起的一种数据结构。闭包允许函数访问在其作用域之外定义的自由变量，并在函数执行时使用它们。当函数

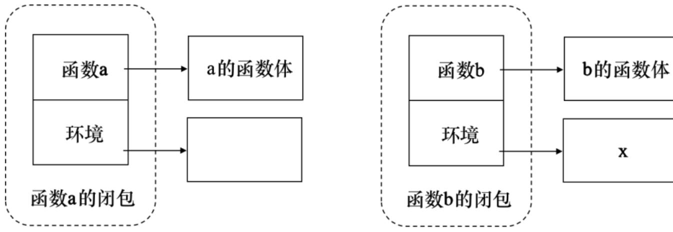


图 2: 函数 a 和 b 的闭包结构

f 被定义时，需要构建该函数 f 的闭包 c，以便后续使用；而当函数（闭包）f 被调用时，需要将闭包中的函数指针 c 和环境 e 分别取出，并将环境 e 作为额外参数，对函数指针 c 进行调用。

以上述讨论的函数 `a` 及其内部嵌套的函数 `b` 为例，它们的闭包结构如图 2所示。函数 `a` 的闭包是指向一个结构体的指针，该结构体包括两个指针字段，分别指向函数 `a` 的内存地址以及函数 `a` 的环境。由于函数 `a` 不包括自由变量，因此其环境为空。而对于函数 `b`，其环境中包括一个自由变量 `x`。

C. 闭包变换算法

闭包变换（Closure conversion）通过将程序中的函数转换为闭包，从而将高阶嵌套函数编译为普通函数。闭包变换算法主要分为 5 个关键步骤：

1) 自由变量分析：通过静态程序分析，得到每个函数的自由变量列表，即环境；
2) 函数闭合：对每个函数定义加入显式的环境参数 e，使得函数闭合；
3) 闭包创建：在函数的每个定义点后，插入为该函数构建闭包的指令；
4) 闭包调用：在函数的每个调用点，插入打开该闭包并进行调用的指令；
5) 提升：将经过闭包变换后的嵌套函数，都提升到最顶层，最终形成平坦的函数结构。

基于上述关键步骤，算法 1给出了闭包变换的具体算法。算法接收包含高阶嵌套函数的程序 p 作为输入，首先调用程序分析算法 freeVarAnalysis()，计算并得到程序 p 中每个函数 f 对应的自由变量列表，并将其存入字典 freeVars 中（即该字典以函数名 f 为关键字，以该函数中的自由变量列表为值）（第 5 行）。

第二步，算法将每个函数闭合（第 6 到 13 行）；具体的，对于函数定义 $f(a_1, \ldots, a_n)$，算法首先给它增加一个参数 env，作为新的第一个形式参数，并且在原来的函数体 S 前，插入 n 条变量读取语句

$$x_i = env[i-1]$$

其中 x_i，$1 \le i \le n$ 是函数 f 的 n 个自由变量。函数闭合操作使得每个函数 f 都变成闭合函数，即不再包括自由变量。

第三步，算法进行闭包创建（第 14 到 18 行），即对程序 p 中每个函数的定义点 f，在其后插入两条新的闭包创建语句，第一条语句根据函数 f 的自由变量，创建一个环境 env（在典型的实现中，可以用任意合适的数据结构，如数组或者链表来作为环境的具体实现），

算法 1：闭包变换

输入： p：包含高阶嵌套函数的程序

输出： p'：完成闭包变换的结果程序

```
1:  // 将每个函数 f 映射到其自由变量列表
2:  freeVars = [ ]
3:  procedure CLSOURECONVERSION(p)
4:      // #1: 自由变量分析
5:      freeVarAnalysis(p)
6:      // #2: 函数闭合
7:      for 每个函数定义 f(a1, ..., an) {S} do
8:          ▷ 将其修改为（f 的自由变量为 x1, ..., xn）:
9:          f(env, a1, ..., an){
10:           x1 = env[0];
11:           ...
12:           xn = env[n − 1];
13:           S}
14:     // #3: 闭包创建
15:     for 函数 f 的每个定义点 do
16:                                 ▷ 在定义点后插入两条语句:
17:         env = newEnv(freeVars[f]);
18:         f = newClosure(f, env);
19:     // #4: 闭包使用
20:     for 每个函数调用 f(a1, ..., an) 进行改写 do
21:         #code(f)(#env(f), a1, ..., an)       ▷ 改写后
22:     // #5: 提升
23:     p′ = lift(经过上述一些列变形后的程序 p)
24:     return p′
```

接着用该环境 env 以及函数的代码指针 f，创建一个闭包 f。这样，程序 p 中所有的函数值 f 都变成了闭包值。

第四步，算法实现闭包调用（第 19 到 21 行），即对每个函数调用点 $f(a_1, \ldots, a_n)$，由于函数 f 都是闭包，算法将该函数调用改写为闭包调用

$$\#code(f)(\#env(f), a_1, \ldots, a_n)$$

其中 $\#code(f)$ 和 $\#env(f)$ 分别表示从闭包 f 中取出其包含的函数指针 $code$ 以及环境 env（参考图 2）。

最后，算法对经过闭包变换的程序进行提升（第 22 到 23 行），即把所有的嵌套函数都提升到最顶层；由于闭包变换已经将所有的函数闭合，函数提升不影响函数词法作用域。

我们将算法 1作用于图 1中的示例程序。首先，算法进行自由变量分析，计算得到函数 a、b 和 c 的自由变量分别为 $[]$、$[x]$ 和 $[x, y]$。

接下来，算法对函数进行闭合，得到

```
a(int[] env, int x){
  b(int[] env, int y){
    x = env[0];
    c(int[] env, int z){
      x = env[0];
      y = env[1];
      return x + y + z; }
    return c; }
  return b;
}
f = a(3);
g = f(4);
h = g(5);
```

经过这个步骤，原本包含自由变量的函数（如函数 b 和 c）已经闭合。

算法继续对上述程序进行闭包创建操作，得到

```
a(int[] env, int x){
  b(int[] env, int y){
    x = env[0];
    c(int[] env, int z){
      x = env[0];
      y = env[1];
      return x + y + z; }
    env = newEnv([x, y]);
    c = newClosure(c, env);
    return c; }
  env = newEnv([x]);
  b = newClosure(b, env);
  return b;
}
env = newEnv([]);
a = newClosure(a, env);
f = a(3);
g = f(4);
h = g(5);
```

经过这个步骤后，程序中的所有函数值（如 a、b、c 等）都被转换为闭包。

接下来，算法对闭包进行调用，将程序变换为

```
a(int[] env, int x){
  b(int[] env, int y){
    x = env[0];
    c(int[] env, int z){
      x = env[0];
      y = env[1];
      return x + y + z; }
    env = newEnv([x, y]);
    c = newClosure(c, env);
    return c; }
  env = newEnv([x]);
  b = newClosure(b, env);
  return b;
}
env = newEnv([]);
a = newClosure(a, env);
f = #code(a)(#env(a), 3);
g = #code(f)(#env(f), 4);
h = #code(g)(#env(g), 5);
```

最后，算法对上述程序代码进行提升，得到

```
c(int[] env, int z){
  x = env[0];
  y = env[1];
  return x + y + z;
}
b(int[] env, int y){
  x = env[0];
  env = newEnv([x, y]);
  c = newClosure(c, env);
  return c;
}
```

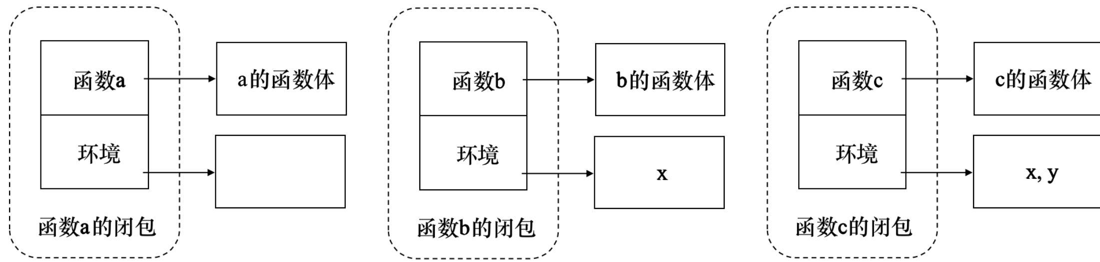


图 3: 平坦闭包示例

```
a(int[] env, int x){
  env = newEnv([x]);
  b = newClosure(b, env);
  return b;
}
env = newEnv([]);
a = newClosure(a, env);
f = #code(a)(#env(a), 3);
g = #code(f)(#env(f), 4);
h = #code(g)(#env(g), 5);
```

最终，该包含高阶嵌套函数的程序，经过闭包变换后，高阶嵌套函数被消除，程序被编译为具有平坦结构的类 C 语言的程序；因此，可以使用本书前面章节讨论的编译算法，将其进一步编译为目标程序。

D. 平坦闭包与链式闭包

根据闭包的具体数据结构组织和实现，闭包可以分为*平坦闭包*（Flat closure）和*链式闭包*（Linked closure）两种形式。在实际编译器实现中，要从编程语言的特性、存储空间、和存取效率等各维度，综合进行闭包的设计和实现决策。

在平坦闭包中，函数的自由变量的值被存储在一个平坦的结构中，这个结构通常是一个向量（或数组），每个自由变量的值都存储在向量中的一个特定位置上。在函数调用时，可以通过索引向量来获取自由变量的值。例如，考虑图 1中的代码示例，函数 `a`、`b` 和 `c` 的平坦闭包结构如图 3所示；需要特别注意，平坦闭包维护了同一个自由变量的独立版本，例如，函数 `b` 和 `c` 同时维护了自由变量 `x`，且这两个 `x` 并不是同一个版本，

即函数 `c` 的闭包把函数 `b` 的闭包中的变量 `x`，拷贝到了自身环境中。

在平坦闭包中，由于每个函数都有自由变量的独立拷贝，因此，函数对自由变量的访问的运行时间复杂度为 $O(1)$，访问执行效率较高。但是，由于闭包中的自由变量被拷贝，闭包中的环境将占据更多的存储空间，一般的，对于存在 n 层嵌套的程序代码，平坦闭包将占据最多 $O(n^2)$ 的存储空间；我们把对这个结论的证明作为练习留给读者完成。

链式闭包（也称为*环境链式闭包*）是指在闭包的环境中既存储了来自直接嵌套函数的自由变量，同时还存储了指向直接嵌套函数环境的指针 `link`。这样，程序既可以通过直接访问其环境来获取来自直接嵌套函数的自由变量，还可以顺着环境指针 `link`，访问位于更外层嵌套函数的自由变量。由于每个闭包的环境都有一个指向其外部嵌套函数的环境指针，因此形成了一个链式结构，称为环境链。例如，对于图 1中的示例代码，其链式闭包的结构如图 4所示。以函数 `c` 为例，如果该函数访问自由变量 `y`，可以从其自身环境中直接得到（因为变量 `y` 处于其直接外部嵌套函数 `b` 中）；而如果函数 `c` 要访问其自由变量 `x`，则从其环境的第一个字段，取出指向其外部环境（即函数 `b` 的环境）的指针 `link`，并从该指针访问函数 `b` 的环境中的变量 `x`。由于在环境构建过程中，算法不需要进行自由变量的拷贝操作，因此，链式闭包的构建效率较高，其最坏执行时间复杂度为 $O(1)$。同时，链式闭包更加节省存储空间：一般的，对于有 n 层嵌套的高阶函数，其存储空间占用在最坏情况下是 $O(n)$。但是，由于函数在访问自由变量时，需要反复访问环境链，因此，对于自由变量 x，其变量访问的最坏运行时间复杂度是 $O(n)$，其中 n 是变量 x 的嵌套深度。，我们把对这些结论的验证，作为练习留给读者。

最后，我们要指出的是，算法 1给出的是基于平坦闭包的构造算法；我们可以把这个算法改造成为基于链式表示的闭包构造算法，我们把算法的实现细节作为习题，留给读者完成。

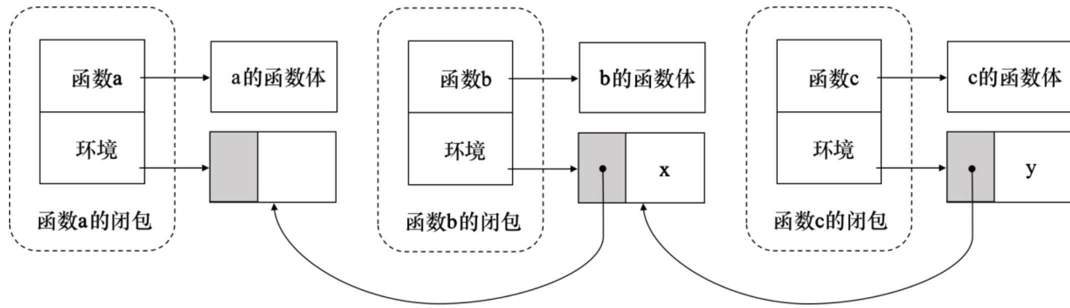


图 4: 链式闭包示例

E. 闭包与静态链

闭包变换算法在中间表示层面对程序进行变形，消除了嵌套高阶函数带来的复杂性，从而简化了编译的后续阶段。在更简单的编译实现中，可以直接基于静态链实现闭包，其是链式闭包的一种特殊形式。

基于静态链的闭包实现包括四个主要步骤：

1) 逃逸分析（escape analysis）：分析当前函数中，会被内层嵌套函数引用的变量，这些变量称为逃逸变量；
2) 栈帧的堆分配：将每个函数的栈帧分成两个部分：其中一部分称为栈栈帧（stack frame），位于调用栈上，用于存放函数的返回地址、被调用者保存寄存器、溢出的变量等；另一部分位于堆中，称为堆栈帧（heap frame），用于存放逃逸变量、该栈帧的静态链等；
3) 闭包构建：对每个函数定义，构建其闭包，其闭包的环境即其直接外层函数的堆栈帧；
4) 闭包调用：对每个函数调用，从闭包中取出函数指针以及环境，将环境作为静态链传递给函数指针，完成调用。

考虑图 1给出的高阶嵌套函数，在函数 `a` 刚被调用、以及在返回前的栈帧布局如图 5所示。在函数 `a` 刚

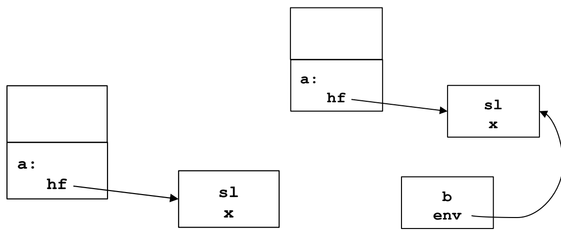


图 5: 函数 `a` 刚被调用以及刚要返回前的栈帧布局

被调用时，为其创建了两个函数帧：栈帧上存放了堆帧的指针 `hf`，堆帧上放置了逃逸变量 `x` 以及静态链 `sl`。在函数 `a` 返回前，为函数指针 `b` 创建了闭包，其中的环境指针 `env` 指向了其外层函数的堆帧。

接着，程序继续调用 `b` 的闭包，`b` 的环境指针作为静态链参数变量传递给 `b` 的代码，形成的帧结构如图 6左图所示。函数 `b` 刚开始执行时，其堆栈帧中包

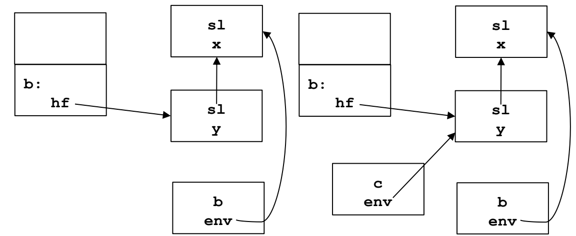


图 6: 函数 `b` 刚被调用以及刚要返回前的栈帧布局

括其逃逸变量 `y`、以及其静态链指针 `sl` 指向其外层函数（即函数 `a`）的堆帧。如果此时函数 `b` 需要访问其自由变量 `x` 的话，可以顺着静态链访问到函数 `a` 的堆帧。在函数 `b` 返回前，它构建了函数 `c` 的闭包，其环境指针 `env` 指向其直接外部函数 `b` 的堆帧。

III. 延续

延续（Continuation）是编程语言中的重要概念，也是函数式语言编译的重要技术。本小节主要讨论延续及其在编译器实现中的作用，主要内容包括延续的基本概念、延续传递风格、延续变换等。

A. 延续的基本概念

延续不仅是一个具体的语言机制，同时也是一个一般的程序设计概念；直观上，延续指的是一个计算中尚未执行的部分或者说是未来的计算，也可以理解为当前计算的“剩余部分”。

为了更直观的理解延续的基本概念，我们考虑一个简单的表达式

$$((3+4)+5)+6$$

这个表达式首先计算子表达式 $(3+4)$，在计算过程中，该计算的剩余部分为 $(\square+5)+6$，即该剩余部分将先计算 $\square$ 和 5 的和，再加上 6；其中 $\square$ 代表正在进行的计算（在当前例子中，即对子表达式 $(3+4)$ 的计算）会返回的结果。同理，子表达式 $((3+4)+5)$ 的剩余计算是 $\square+6$。

“剩余计算”的概念可以非常精确的被表达成函数的形式。例如，上述计算可以被写成

```
k(x){
  h(y){ y + 6; }
  u = x + 5; h(u)
}
v = 3 + 4; k(v)
```

其中子表达式 $(3+4)$ 的剩余计算被表达为一个函数 `k`，子表达式 $3+4$ 的计算结果 v，作为参数传递给该函数；而该函数 `k` 负责完成剩余的计算 $x+5$。同理，子表达式 $((3+4)+5)$ 的剩余计算是 $\square+6$ 被表达成函数 `h`，它负责完成整个表达式的计算。

延续不仅是一种控制结构，还给定了被编译程序的一种重要的中间表示；其中，所有的复合计算都被分解为简单形式，并且，它还显式指定了程序的执行流程。例如，对于上述程序，实际被执行的计算也可写成类似三地址码的形式：

```
v = 3 + 4;
x = v;
u = x + 5;
y = u;
y + 6;
```

其不但给出了计算的具体形式，还给定了计算执行的顺序。但是，延续形式和三地址形式有一个显著的不同，即在三地址码形式中，计算的执行顺序是隐式的，即计算以语句排列的先后进行；而在延续形式中，计算的执行顺序是显式的，取决于延续调用的先后顺序。这个性质也意味着在三地址码上更方便进行需要代码重排的优化；例如，在指令调度时，可以根据不同语句间的数据依赖关系，对三地址码中的指令进行重排。

B. 延续传递风格

延续传递风格（Continuation-Passing Style，CPS）指的是一种特定的编程风格，在这种编程风格中，每个函数除接受原有的参数外，还接受其延续作为额外的参数。这个延续函数参数接受该函数的返回值，并指明了当前函数在返回后需要继续执行的操作。这样，在当前函数返回时，它将自己的返回值作为参数，传递给其延续参数，从而继续执行剩余的计算。

考虑上一小节的示例程序，用延续传递风格，可将其写成：

```
add(k, x, y){ k(x+y); }
k(x){
  h(y){ add(halt, y, 6); }
  add(h, x, 5);
}
add(k, 3, 4);
```

其中加法函数 `add` 除了接受被加数和加数 x 和 y 之外，还接受额外的延续参数 k，加法函数不返回，而是把其计算结果 $x+y$ 直接作为参数，调用延续 $k(x+y)$。在该函数的调用点，需要将当前计算的延续作为参数，显式传递给该函数。例如，在调用 `add(k, 3, 4)` 时，将延续 $k(x)\{\ldots\}$ 传递给了该调用。

延续传递风格可以用于实现任意的函数，甚至包括递归函数。例如，考虑下面递归版本的求和程序：

```
sum(n){
  if(n==0) return 0;
  else return n + sum(n-1); }
```

它可被写成如下的延续传递风格的程序：

```
sum(k, n){
  if(n==0) k(0);
  else{
    h(y){ k(n+y); }
    sum(h, n-1); }
}
```

可以看到，尽管延续传递风格可使得函数具有统一的形式，但是，由于其控制流中涉及复杂的函数回调关系，手工构造具有延续传递风格的程序并不容易；因此，我们需要一种系统的方法来自动将任意程序转换为延续传递风格。

C. 延续传递风格变换

延续传递风格变换（CPS conversion）是一种系统性的算法，将任意风格的程序，转换为等价的但使用延续传递风格的程序。延续传递风格变换的算法，依赖于具体被变换的语言；在本小节，我们首先讨论对于一个小的命令式语言的转换；在下一个小节，我们讨论在一个函数式语言上的延续传递风格变换。

我们首先定义一个小的类 C 的命令式语言 MiniC，其语法规则如下：

表达式	b	$::=$	$x == n \mid x <= n \mid x < n \mid b \| b \mid \ldots$
语句	s	$::=$	$x = n \mid x = y \mid x = y \oplus z \mid y = f(x)$
		$\mid$	$\texttt{if}(b)\ \vec{s}\ \texttt{else}\ \vec{s} \mid \texttt{return}\ x$
函数	f	$::=$	$y(x)\{\vec{s}\}$
程序	p	$::=$	$\vec{f}$

布尔表达式 b 计算得到布尔值，既包括整型数值上的“相等”等大小比较，也包括布尔值上的或“$\|$”等布尔运算。

语句 s 包括六种可能的形式：常量赋值 $x=n$、变量赋值 $x=y$、二元运算 $x=y\oplus z$、函数调用 $y=f(x)$、条件判断 $\texttt{if0}(x)\ \vec{s_1}\ \texttt{else}\ \vec{s_2}$、和返回语句 $\texttt{return}\ x$。这里，符号 $\vec{s}$ 代表由语句 s 组成的有序列表，特别的，如果列表为空，我们记为 $\cdot$。

函数 f 除了包括函数名 y 以及参数 x 外，还包括函数体 $\vec{s}$。

程序 p 由函数列表 $\vec{f}$ 组成。

对 MiniC 中语句列表 $\vec{s}$，其延续变换规则的形式是

$$[\![\vec{s}]\!]_j^k$$

其中 k 是当前函数接受的延续参数变量，而 j 是在变换过程中引入的局部延续变量。

根据语句列表 $\vec{s}$ 转换的形式，基于对列表 $\vec{s}$ 中第一条语句语法形式的归纳，我们给出如下的转换规则：

$$
\begin{aligned}
[\![x=n;\vec{s}]\!]_j^k &= x=n;[\![\vec{s}]\!]_j^k \\
[\![x=y;\vec{s}]\!]_j^k &= x=y;[\![\vec{s}]\!]_j^k \\
[\![x=y\oplus z;\vec{s}]\!]_j^k &= x=y\oplus z;[\![\vec{s}]\!]_j^k \\
[\![y=f(x);\vec{s}]\!]_j^k &= h(a)\{y=a;[\![\vec{s}]\!]_j^k\} \\
&\quad f(h,x); \\
[\![\texttt{if}(b)\ \vec{s_1}\ \texttt{else}\ \vec{s_2};\vec{s}]\!]_j^k &= h(a)\{[\![\vec{s}]\!]_j^k\} \\
&\quad \texttt{if}(b)\ [\![\vec{s_1}]\!]_h^k\ \texttt{else}\ [\![\vec{s_2}]\!]_h^k \\
[\![\texttt{return}\ x;\vec{s}]\!]_j^k &= k(x) \\
[\![\cdot]\!]_j^k &= j(0)
\end{aligned}
$$

其中，若语句列表以三种赋值语句开始（对应前三条规则），则赋值语句保持不变，并继续转换剩余的语句。

对于函数调用 $y=f(x)$，规则新生成一个延续 $h(a)\{y=a;[\![\vec{s}]\!]_j^k\}$，其函数体既包括对变量 y 的赋值 $y=a$，也包括对剩余语句列表 $\vec{s}$ 的转换 $\vec{s}_j^k$；函数调用加入了额外的延续参数 h 后，成为 $f(h,x)$。

对于条件语句 $\texttt{if}(b)\ \vec{s_1}\ \texttt{else}\ \vec{s_2}$，规则首先生成一个新的延续 $h(a)\{[\![\vec{s}]\!]_j^k\}$，其中包括对剩余语句 s 的转换结果。同时，规则用新生成的延续 h，对条件语句的两个分支 $\vec{s_1}$ 和 $\vec{s_2}$ 进行转换。需要注意的是，从技术上，延续 h 并不需要接受参数，因为该延续并不需要条件分支的值，但是，为了跟延续 k 保持一致，我们令其接受一个平凡的值 a。

对于返回语句 $\texttt{return}\ x$，规则直接将返回值 x 传递给延续 k；并且由于返回语句后的列表 $\vec{s}$ 不会被执行，可以直接省略。

最后，对空语句列表 $\cdot$，规则将直接调用局部延续 j 并传入一个平凡的参数值 0，这意味着所有语句已经被转换完，可以跳转到该延续继续执行。

对于函数定义 f，转换规则

$$[\![y(x)\{\vec{s}\}]\!] = y(k,x)\{[\![s]\!]_k^k\}$$

给函数 y 添加新的闭包参数 k，并继续转换其函数体 $[\![s]\!]_k^k$，其中全局延续和局部延续都设置成 k。

对于整个程序 p，规则

$$[\![f_1\ldots f_n]\!] = [\![f_1]\!]\ldots[\![f_n]\!]$$

将分别转换每个函数 f_i，$1\le i\le n$。

接下来，我们结合具体实例，研究上述延续传递风格变换规则的应用。首先，考虑如下的求绝对值的函数

```
abs(n){
  if(n<0) y = 0-n;
  else y = n;
  return y;
}
```

延续传递风格变换规则可将其转换为如下程序：

```
abs(k, n){
  h(a){ k(y); }
  if(n<0){ y = 0-n; h(0); }
  else{ y = n; h(0); }
}
```

可以看到，对于条件语句，延续传递风格变换得到的结果程序，其结构非常接近其控制流图结构。

对于前一个小节中我们讨论的递归求和函数 `sum`，延续传递风格变换规则可将其转换为如下程序：

```
sum(k, n){
  h1(a){ k(0); }
  if(n==0)
    k(0);
  else{
    h2(a){
      y = n + a;
      k(y);
    }
    sum(h2, n - 1);
  }
}
```

我们把对这个程序具体的转换过程，作为练习留给读者完成。特别需要注意到，对条件语句的转换生成了一个无用的延续 `h1`，其可被后续的死代码删除优化移除。

接下来，我们考虑一个更加复杂的例子。如下给定了基于递归实现的斐波那契数的计算函数 `fib`：

```
fib(n){
  if(n <= 1)
    return 1;
  else{
    f1 = fib(n - 2);
    f2 = fib(n - 1);
    z = f1 + f2;
    return z;
  }
}
```

注意到，`else` 分支中的代码实际上等价于 `return fib(n-2)+fib(n-1);` 但目前的语法结构更利于进行延续传递风格的代码变换；并且，我们允许函数 `fib` 接受一个表达式作为参数，这不影响转换的进行以及结果。对该函数进行变换后的程序如下所示：

```
fib(k, n){
  h1(a) { k(a); }
  if(n <= 1)
    k(1);
  else{
    h2(a){
      f1 = a;
      h3(b){
        f2 = b;
        z = f1 + f2;
        k(z);
      }
      fib(h3, n-1);
    }
    fib(h2, n - 2);
  }
}
```

我们同样把对该函数的延续传递风格转换的具体过程，作为练习留给读者完成。

IV. λ 演算的编译

本小节以对 λ 演算的编译为例，讨论一个简单的函数式语言的编译器，其编译阶段包括延续变换、闭包变换，并最终生成 RISC-V 目标代码。

A. λ 演算

首先，我们给出支持整型值的简单无类型 λ 演算，其语法如下所示：

$$\text{表达式} \quad e \quad ::= \quad n \mid x \mid e + e \mid \lambda x.e \mid e\ e$$

其中的非终结符 e 表示表达式，包括五种语法形式：符号 n 表示整型值常量；符号 x 表示表达式变量；表达式 $e + e$ 表示加法；表达式 $\lambda x.e$ 表示函数抽象；式 $e\ e$ 表示函数调用。

需要注意，λ 演算是一种无名形式的函数定义和函数调用，例如，图 1中的示例程序，用 λ 演算可表示成：

$(\lambda x.\lambda y.\lambda z.(x+y+z))$ 3 4 5

函数调用默认左结合。

B. 延续传递风格变换

在给出 λ 演算的延续变换规则前，我们首先给出变换的目标语言的定义，我们称该语言为 λ^P，其语法规则如下所示。

$$
\begin{array}{llll}
\text{表达式} & e & ::= & \texttt{let } x = n \texttt{ in } e \\
& & | & \texttt{let } x = y + z \texttt{ in } e \\
& & | & \texttt{let } f\ h\ x = e \\
& & | & \texttt{letcont } k\ x = e \\
& & | & x\ k\ z \\
& & | & k\ x
\end{array}
$$

其中前两项定义了整型变量和整型上的二元运算（即加法），其中的 let 语法形式给出了变量的显式词法作用域；第三项语法定义了函数抽象，其中函数具有显式的名字 f，并且接受两个参数 h 和 x，其中参数 h 是延续参数，并且该函数的体是 e；第四个语法 letcont 定义了一个延续函数 k，其具有一个参数 x，函数体是 e；最后两个语法形式 $x\ k\ z$ 和 $k\ x$ 分别是对普通函数的调用，以及对延续函数的调用。

对于 λ 演算，我们给出其延续变换的翻译规则，规则的一般形式是：

$$[\![e]\!]k : \lambda \to \lambda^P$$

即 $[\![e]\!]k$ 将 λ 演算中的表达式 e，翻译到 λ^P 演算中的表达式；其中 k 是一个延续变换的辅助函数，具有类型 $x \to \lambda^P$，即该函数将表达式 e 翻译得到的结果变量 x，映射到 λ^P 的结果表达式。

翻译规则 $[\![e]\!]k$ 基于对表达式 e 的语法形式归纳给出：

$$
\begin{aligned}
[\![n]\!]k &= \texttt{let } x = n \texttt{ in } k(x) \\
[\![x]\!]k &= k(x) \\
[\![e_1 + e_2]\!]k &= [\![e_1]\!](\lambda x_1.[\![e_2]\!](\lambda x_2.\texttt{let } x = x_1 + x_2 \texttt{ in } k(x))) \\
[\![\lambda x.e]\!]k &= \texttt{let } f\ h\ x = ([\![e]\!](\lambda z.h\ z)) \texttt{ in } k(f) \\
[\![e_1\ e_2]\!]k &= [\![e_1]\!](\lambda x_1.([\![e_2]\!]\lambda x_2.(\texttt{letcont } h\ x = k\ x \texttt{ in } x_1\ h\ x_2)))
\end{aligned}
$$

对于整型常量 n，规则引入了辅助变量 x，并将其传递给延续函数 k。对于程序变量 x，规则将其直接传递给延续函数 k。对于加法表达式 e_1+e_2，规则首先递归翻译子表达式 e_1 和 e_2，分别得到结果 x_1 和 x_2，再将二者的和 x_1+x_2 赋值给变量 x 后，将变量 x 传递给延续函数 k。对于函数抽象 $\lambda x.e$，规则首先创建一个被显式命名为 f 的函数（即 f 是一个全新的变量名），并将其传递给延续函数 k，该函数 f 接受一个额外的延续参数 h，接着，规则用新的延续函数 $\lambda z.h\ z$ 递归翻译原来的函数体 e。对于函数调用 $e_1\ e_2$，规则首先分别递归翻译两个子表达式 e_1 和 e_2，得到两个结果 x_1 和 x_2，然后生成新的函数调用 $x_1\ h\ x_2$，其中 h 是被 letcont 定义的一个新的延续函数。

作为示例，我们研究上述规则在表达式

$$(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4\ 5$$

上的翻译结果。

首先，我们用延续函数 $\lambda x.\texttt{halt}(x)$ 翻译整个表达式，其中 halt 是一个预置的延续函数，接受程序最终的值。规则首先翻译最外层的函数调用，得到如下结果：

$$
\begin{aligned}
&[\![(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4\ 5]\!](\lambda x.\texttt{halt}(x)) \\
= \ &[\![(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4]\!] \\
&\quad (\lambda a_1.[\![5]\!](\lambda a_2.(\texttt{letcont } a_3\ x_1 = \texttt{halt}(x_1) \texttt{ in } a_1\ a_3\ a_2))) \\
= \ &[\![(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4]\!] \\
&\quad (\lambda a_1.(\texttt{letcont } a_3\ x_1 = \texttt{halt}(x_1) \\
&\qquad \texttt{in } a_1\ a_3\ 5))
\end{aligned}
$$

需要注意，为清晰起见，我们对最后一个延续调用做了常量传播优化，将常量 5 传播到延续调用 $a_1\ a_3\ 5$ 中。

类似的，我们接下来应用延续变换规则，对表达式继续进行翻译，得到：

$$
\begin{aligned}
= \ &[\![(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3]\!] \\
&\quad (\lambda b_1.(\texttt{letcont } b_3\ y_1 = \\
&\quad\ \texttt{letcont } a_3\ x_1 = \texttt{halt}(x_1) \\
&\qquad \texttt{in } y_1\ a_3\ 5)) \\
&\quad \texttt{in } b_1\ b_3\ 4)) \\
= \ &[\![\lambda x.\lambda y.\lambda z.(x+y+z)]\!] \\
&\quad \lambda c_1.(\texttt{letcont } c_3\ z_1 = \\
&\qquad \texttt{letcont } b_3\ y_1 = \\
&\qquad\quad \texttt{letcont } a_3\ x_1 = \texttt{halt}(x_1) \\
&\qquad\quad \texttt{in } y_1\ a_3\ 5 \\
&\qquad \texttt{in } z_1\ b_3\ 4 \\
&\quad \texttt{in } c_1\ c_3\ 3)
\end{aligned}
$$

记上式最后的延续函数为 k'，则上式可进一步计算得到：

```
= let d1 d2 x = [[λy.λz.(x + y + z)]](λd.d2 d)
  in letcont c3 z1 =
        letcont b3 y1 =
            letcont a3 x1 = halt(x1)
            in y1 a3 5
        in z1 b3 4
     in d1 c3 3
= let d1 d2 x =
      let e1 e2 y = [[λz.(x + y + z)]](λe.e2 e)
      in d2 e1
  in letcont c3 z1 =
         letcont b3 y1 =
             letcont a3 x1 = halt(x1)
              in y1 a3 5
         in z1 b3 4
      in d1 c3 3
= let d1 d2 x =
      let e1 e2 y =
              let f1 f2 z = [[x + y + z]](λf.f2 f)
              in  e2 f1
      in d2 e1
  in letcont c3 z1 =
         letcont b3 y1 =
             letcont a3 x1 = halt(x1)
              in y1 a3 5
         in z1 b3 4
      in d1 c3 3
= let d1 d2 x =
      let e1 e2 y =
              let f1 f2 z = f2(x + y + z)
              in  e2 f1
      in d2 e1
  in letcont c3 z1 =
         letcont b3 y1 =
             letcont a3 x1 = halt(x1)
              in y1 a3 5
         in z1 b3 4
      in d1 c3 3
```

C. 闭包变换

编译器在利用延续变换规则，生成 λ^P 演算的表达式后，可继续利用闭包变换规则，将函数闭合。我们首先给出闭包变换的目标语言的定义，我们称其为 λ^C，其语法规则如下所示。

$$
\begin{array}{llll}
\text{表达式} & e & ::= & \texttt{let}\ x = n\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y + z\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y[n]\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = \texttt{makeClosure}(f, [\ldots])\ \texttt{in}\ e \\
 & & | & \texttt{let}\ f(v, h, x) = e \\
 & & | & \texttt{letcont}\ k(v, x) = e \\
 & & | & f(k, x) \\
 & & | & k(x)
\end{array}
$$

和演算 λ^P 中的表达式相比，演算 λ^C 中的表达式中增加了两种形式：语法形式 $y[n]$ 代表从环境 y 中取得第 n 个分量；而语法形式 $\texttt{makeClosure}(f, [\ldots])$ 表示用给定的函数指针 f 以及自由变量列表构建闭包。

除了新增加的五种语法，还有四种语法形式（即最后的四种）发生了变化：其中，函数 $\texttt{let}\ f(v, h, x) = e$ 和延续 $\texttt{letcont}\ k(v, x) = e$ 的定义都分别增加了额外的第一参数 v 表示环境，并且改成了非 Curry 形式；而函数调用 $f(k, x)$ 或延续调用 $k(x)$ 都代表了对闭包的调用，因此不需要显式的环境参数。

闭包变换可同样采用算法 1进行，即分成五个步骤进行，篇幅所限，此处不再赘述该算法。对于前面讨论的示例程序，进行完算法 1的前四个步骤后，得到如下结果程序（我们将在下个小节单独讨论第五个步骤）：

```
let d1(env, d2, x) =
  let e1(env, e2, y) =
              let x = env[0] in
              let f1(env, f2, z) =
                  let x = env[0] in
                  let y = env[1] in
                  f2(x + y + z)
              in let f1 = makeClosure(f1, [x, y])
                 in  e2(f1)
  in let e1 = makeClosure(e1, [x])
     in d2(e1)
in let d1 = makeClosure(d1, [])
   in letcont c3(env, z1) =
          letcont b3(env, y1) =
              letcont a3(env, x1) = halt(x1)
              in let a3 = makeClosure(a3, [])
                 in y1(a3, 5)
          in let b3 = makeClosure(b3, [])
             in z1(b3, 4)
    in let c3 = makeClosure(c3, [])
```

$\texttt{in}\ d_1(c_3, 3)$

D. 函数提升

函数提升阶段将程序中出现的所有的函数都提升到最顶层，即程序中不再有嵌套函数。为刻画函数提升后的程序的语法规则，我们用如下的语法规则定义一个新语言 λ^L：

$$
\begin{array}{llll}
\text{表达式} & e & ::= & \texttt{let}\ x = n\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y + z\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y[n]\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = \texttt{makeClosure}(f, [\ldots])\ \texttt{in}\ e \\
 & & | & f(k, x) \\
 & & | & k(x) \\
\text{函数} & f & ::= & \texttt{letfun}\ x(y, h, z) = e \\
 & & | & \texttt{letcont}\ k(y, x) = e \\
\text{程序} & p & ::= & \texttt{let}\ f_1\ \ldots\ f_n\ \texttt{in}\ e
\end{array}
$$

程序 p 包括 n 个函数定义 $f_1\ \ldots\ f_n$，后跟一个表达式 e，这个 n 个函数可以相互递归调用，并在表达式 e 中有效。函数 f 包括普通函数和延续函数，其中普通函数用 `letfun` 定义。

对于上述示例程序，进一步经过函数提升后，将得到：

```
let
letfun d1(env, d2, x) =
  let e1 = makeClosure(e1, [x]) in
  d2(e1)
letfun e1(env, e2, y) =
  let x = env[0] in
  let f1 = makeClosure(f1, [x, y]) in
  e2(f1)
letfun f1(env, f2, z) =
  let x = env[0] in
  let y = env[1] in
  f2(x + y + z)
letcont a3(env, x1) =
  halt(x1)
letcont b3(env, y1) =
  let a3 = makeClosure(a3, []) in
  y1(a3, 5)
letcont c3(env, z1) =
  let b3 = makeClosure(b3, []) in
  z1(b3, 4)
in
  let d1 = makeClosure(d1, []) in
  let c3 = makeClosure(c3, []) in
  d1(c3, 3)
```

E. 显式分配

语言 λ^L 中的程序仍然包括隐式的环境构建及访问，以及闭包构建及访问的操作，显式分配阶段将内存的分配即访问的操作显式化，从而方便后续的代码生成。为此，我们定义如下的 λ^A 语言：

$$
\begin{array}{llll}
\text{表达式} & e & ::= & \texttt{let}\ x = n\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y + z\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = y[n]\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x = \texttt{alloc}(n)\ \texttt{in}\ e \\
 & & | & \texttt{let}\ x[n] = y\ \texttt{in}\ e \\
 & & | & f(v, k, x) \\
 & & | & k(v, x) \\
\text{函数} & f & ::= & \texttt{letfun}\ x(v, h, y) = e \\
 & & | & \texttt{letcont}\ k(v, x) = e \\
\text{程序} & p & ::= & \texttt{let}\ f_1\ \ldots\ f_n\ \texttt{in}\ e
\end{array}
$$

表达式 e 新增了两种语法形式：语法 $\texttt{let}\ x = \texttt{alloc}(n)\ \texttt{in}\ e$ 分配 n 个内存单元，并将分配得到的内存指针赋值给变量 x；语法 $\texttt{let}\ x[n] = y\ \texttt{in}\ e$ 将变量 y 的值，赋值给变量 x 所指向内存单元的第 n 个分量。需要注意，函数 `alloc` 是一个内置的库函数，它接受要分配的内存单元个数作为参数，而每个内存单元的大小依赖于编译的具体目标机器的字长。

表达式 e 中的函数调用 $f(v, k, x)$ 或延续调用 $k(v, x)$ 的语法形式也发生了变化，即都增加了显式的环境参数 v。

我们给出如下的从语言 λ^L 到 λ^A 的编译规则

$$T : \lambda^L \to \lambda^A$$

简单起见，下面只给出对闭包构建和闭包调用的规则，我们把其它语法结构的编译规则作为练习，留给读者完成。

```
T(let x = makeClosure(f, [y1, ..., yn]) in e) =
    let p = alloc(n) in
    let p[0] = y1 in
    ...
    let p[n-1] = yn in
```

```
    let x = alloc(2) in
    let x[0] = f in
    let x[1] = p in
    T(e)
T(f(k, x)) =
    let c = f[0] in
    let v = f[1] in
    c(v, k, x)
T(k(x)) =
    let c = k[0] in
    let v = k[1] in
    c(v, x)
```

对于上述示例程序，进一步经过函数提升后，我们给出上述示例程序中函数 e_1 的编译结果，把剩余程序的编译结果留给读者完成。

```
letfun e1(env, e2, y) =
  let x = env[0] in
  let p = alloc(2) in
  let p[0] = x in
  let p[1] = y in
  let f1 = alloc(2) in
  let f1[0] = f1 in
  let f1[1] = p in
  let c = e2[0] in
  let v = e2[1] in
  c(v, f1)
```

F. RISC-V 代码生成

编译器为 λ^A 的程序生成代码，我们选择 RISC-V 作为代码生成的目标指令集体系结构。RISC-V 是目前最重要的开源精简指令集体系结构之一，因此，选择 RISC-V 作为编译目标，有助于研究和揭示在精简指令集体系结构上的一般性代码生成技术。

RISC-V 指令集是一个较大的指令集家族，且拥有许多扩展；并且，同一个指令集也包括丰富的指令。而我们仅需要 RISC-V 指令集 32 位版本 RV32I 的一个子集，我们称该子集为 R32，下面首先给出其定义：

寄存器	r	::=	`x0` | ... | `x31`
指令	i	::=	`li` r, n
		|	`add` r, r, r
		|	`lw` $r, n(r)$
		|	`sw` $r, n(r)$
		|	`call` f
		|	`jalr` r
		|	`mv` r, r
函数	f	::=	`l` : $i_1 \ldots i_n$
程序	p	::=	$f_1 \ \ldots \ f_n$

寄存器 r 包括 32 个 32 位寄存器 `x0` 到 `x31`，为清晰起见，我们也经常使用这些寄存器在调用约定中的别名，例如，参数传递寄存器 `x10` 到 `x17` 在调用约定中分别称为 `a0` 到 `a7`。

指令 i 包括立即数加载指令 `li`、加法指令 `add`、内存读取指令 `lw` 和存入指令 `sw`、函数调用指令 `call` 及 `jalr`、以及寄存器间的数据移动指令 `mv`。

函数 f 以一个指令标号 `l` 开头，后跟 n 条指令 i_1，…，i_n。

程序 p 包括 n 个函数 f_1，…，f_n，其中有一个名为 `main` 的函数作为程序的执行入口。

基于该指令集 R32 的定义，我们给出从语言 λ^A 到 R32 的编译规则：规则 T_e、T_f 和 T_p 分别负责编译 λ^A 中的表达式 e、函数 f 和程序 p。

编译规则 T_e 的定义如下：

$$
\begin{aligned}
T_e(\texttt{let}\ x = n\ \texttt{in}\ e) &= \texttt{li}\ r_x, n \\
&\quad T_e(e) \\
T_e(\texttt{let}\ x = y + z\ \texttt{in}\ e) &= \texttt{add}\ r_x, r_y, r_z \\
&\quad T_e(e) \\
T_e(\texttt{let}\ x = y[n]\ \texttt{in}\ e) &= \texttt{lw}\ r_x, n(r_y) \\
&\quad T_e(e) \\
T_e(\texttt{let}\ x = \texttt{alloc}(n)\ \texttt{in}\ e) &= \texttt{li}\ a_0, n \\
&\quad \texttt{call alloc} \\
&\quad \texttt{mv}\ r_x, a_0 \\
&\quad T_e(e) \\
T_e(\texttt{let}\ x[n] = y\ \texttt{in}\ e) &= \texttt{sw}\ r_y, n(r_x) \\
&\quad T_e(e) \\
T_e(f(v, k, x)) &= \texttt{mv}\ a_0, r_v \\
&\quad \texttt{mv}\ a_1, r_k \\
&\quad \texttt{mv}\ a_2, r_x \\
&\quad \texttt{jalr}\ r_f \\
T_e(k(v, x)) &= \texttt{mv}\ a_0, r_v \\
&\quad \texttt{mv}\ a_1, r_x \\
&\quad \texttt{jalr}\ r_k
\end{aligned}
$$

这些编译规则比较直接，但有三个关键点需要特别注意：一是编译的过程假定完成了寄存器分配，即对于 λ^A 中的程序变量 x，假定其被分配到寄存器 r_x 中；尽管寄存器分配是编译器后端非常重要的阶段，但其具体算法和过程已经在本书前面章节讨论过，故此处从略。第二，编译规则遵守了 RISC-V 的默认函数调用规范，即函数的参数（前 8 个）传递到寄存器 `a0` 到 `a7` 中，而函数返回值放置在寄存器 `a0` 中（如果是两个，则放置在 `a0` 和 `a1` 中）。最后，为清晰起见，编译规则忽略了调用者和被调用者寄存器保存规范，我们将其作为习题，留给读者完成。

编译规则 T_f 的定义如下：

$$
\begin{aligned}
T_f(\texttt{letfun}\ x(v, h, y) = e) &= \texttt{x}: \\
&\quad T_e(e) \\
T_f(\texttt{letcont}\ k(v, x) = e) &= \texttt{k}: \\
&\quad T_e(e)
\end{aligned}
$$

函数名 x 或延续名 k 分别被编译成代码地址 `x` 或 `k`，后跟函数 $T_e()$ 对函数体 e 编译生成的结果 $T_e(e)$。需要注意，函数的形式参数是预着色的寄存器，即默认已经被分配到参数寄存器 `a0` 到 `a7` 中。

编译规则 T_p 的定义如下：

$$
\begin{aligned}
T_p(\texttt{let}\ f_1\ \dots\ f_n\ \texttt{in}\ e) &= T_f(f_1) \\
&\quad \vdots \\
&\quad T_f(f_n) \\
&\quad \texttt{main}: \\
&\quad\quad T_e(e)
\end{aligned}
$$

其中的 n 个函数 f_1 到 T_n 分别被函数 T_f 编译到汇编代码，而表达式 e 被编译函数 T_e 编译为名为 `main` 的主函数，此即整个程序的执行入口。

利用上述编译函数，上述示例程序中的函数 e_1 将被编译成：

```
e1: // env: a0, e2: a1, y: a2
  ld r_x, 0(a0)
  li a0, 2
  call alloc
  mv r_p, a0
  sw r_x, 0(r_p)
  sw r_y, 1(r_p)
  li a0, 2
  call alloc
  mv r_f1, a0
  sw r_f1, 0(r_f1)
  sw r_p, 1(r_f1)
  lw r_c, 0(a1)
  lw r_v, 1(a1)
  mv a0, r_v
  mv a1, r_f1
  jalr r_c
```

其它函数的编译与此类似，我们把它们作为练习留给读者完成。

V. 优化

本书前面讨论的大部分优化，都可以直接用到函数式语言编译器中。例如，在延续传递风格的中间表示上，可以进行常量传播、复写传播或常量折叠优化等；而在后端，可以进行寄存器分配、指令调度等优化。

根据函数式编译器的独特结构特点，部分优化应用于函数式语言编译器中会更加有效。在本小节，我们讨论两种对函数式语言编译器特别有效的优化：内联展开和尾调用优化。

A. 内联展开

在基于延续传递风格的函数式编译器中，编译会生成大量延续，并且，这些延续的代码量一般较小。内联展开（Inline expansion）可以通过将函数定义代码替换到函数的调用点，从而将函数定义及其调用移除，从而有效降低函数调用的开销。

例如，考虑前述讨论过的延续风格变换生成的示例程序：

```
abs(k, n){
  h(a){ k(y); }
  if(n<0){ y = 0-n; h(0); }
  else{ y = n; h(0); }
}
```

通过将延续 h 分别代入到两个调用点，我们将得到：

```
abs(k, n){
  if(n<0){ y = 0-n; k(y); }
  else{ y = n; k(y); }
}
```

延续 h 在被内联展开后，成为死代码，从而可被移除。

算法 2给出了内联展开的具体步骤。算法接收初始程序 p 作为输入，返回经过内联展开后得到的结果程序。算法对所有的函数进行参数的重命名（第 2 到 5 行）；这个步骤是为了防止出现变量捕获（Variable capture），考虑如下示例程序：

```
f(k, x){
  g(y){ k(x + y); }
  h(x){ g(x); }
}
```

如果我们直接将延续 g 进行内联展开，则生成：

```
f(k, x){
  g(y){ k(x + y); }
  h(x){ k(x + x); }
}
```

注意程序第 2 行的变量 x 本来绑定到第 1 行的参数，但经过内联展开后，x 被错误的绑定到第 3 行函数

算法 2： 内联展开

输入： p：内联展开前的程序
输出： p'：完成内联展开的程序

1: **procedure** InlineExpansion(p)
2:   **for** 每个函数定义 $f(x_1, \ldots, x_n)\ \{S\}$ **do**
3:     ▷ 将其进行参数重命名，改写为如下形式
4:     $f(y_1, \ldots, y_n)\ \{S[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]\}$
5:     ▷ 其中 y_i，$1 \le i \le n$ 都是全新变量
6:   **for** 每个函数调用 $f(e_1, \ldots, e_n)$ **do**
7:     ▷ 假设其定义为 $f(x_1, \ldots, x_n)\{S\}$
8:     ▷ 将该调用改写为如下序列
9:     $y_1 = e_1$
10:     $\ldots$
11:     $y_n = e_n$
12:     $S[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$
13:     ▷ 其中 y_i，$1 \le i \le n$ 都是全新变量
14:   **for** 每个函数定义 $f(x_1, \ldots, x_n)\ \{S\}$ **do**
15:     **if** f 是无用代码 **then**
16:       从程序 p 中移除函数 f
17:   **return** p' ▷ 经过内联展开后得到程序 p'

x 的参数，程序语义发生了改变。因此，为防止出现变量捕获，我们可以将所有函数定义，对其参数 x_i，$1 \le i \le n$，进行函数参数的重命名，将其分别重命名为 y_i，$1 \le i \le n$，并且函数体中出现的参数名进行替换 $S[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$。

接着，算法对每个函数调用 $f(e_1, \ldots, e_n)$ 进行内联展开。首先，算法生成 n 个赋值语句 $y_i = e_i$，$1 \le i \le n$，其中 y_i，$1 \le i \le n$ 是全新变量；然后将函数调用 $f(e_1, \ldots, e_n)$ 替换为其函数体 $S[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$。算法首先生成 n 个新的赋值语句的主要作用，是防止出现不应有的副作用（Side effect）。例如，考虑如下的示例程序

```
f(x){ k (x + x); }
f(print(5));
```

其中函数 print() 打印参数后，返回输出的字符个数（即对本例，其将返回 1）；如果直接进行内联展开，则得到

```
f(x){ k (x + x); }
```

```
k(print(5) + print(5));
```

则参数 5 会被错误的打印两遍。而先将函数参数赋值给变量，可以避免额外副作用的出现：

```
f(x){ k (x + x); }
y = print(5);
k(y + y);
```

最后，内联展开可能会产生无用（函数）代码，即该函数没有被调用，也没有被其它形式引用（如当做函数参数传递等）。因此，算法最后尝试将这类无用函数进行移除，从而有效减小代码的规模。

对递归函数的内联展开更加复杂。考虑如下用递归函数实现的求和程序 `sum` （该函数已经经过了延续传递风格变换）：

```
sum(k, x){
  if(x == 0) k(0);
  else{
    f(a) { k (a + x); }
    sum(f, x-1); }
}
g(k, y){
  sum(k, y);
}
```

直接对函数 `sum` 进行内联展开，我们将得到：

```
sum(k, x){
  if(x == 0) k(0);
  else{
    f(a) { k (a + x); }
    sum(f, x-1); }
}
g(k, y){
  if(y == 0) k(0);
  else{
    f(a) { k (a + y); }
    sum(f, y-1); }
}
```

这种直接展开方式会导致一个问题：即由于递归的存在，展开后的代码，仍然会引用原始函数，从而导致展开后的代码并没有闭合（例如，在本例中，展开后的程序仍会调用函数 `sum`）。

理想上，我们希望经过内联展开的程序包含被内联函数的一个独立版本；为此，我们可使用*循环头变换*（Loop preheader transformation）。如下所示，该变换的基本思想是将递归函数 f 分拆为两个独立函数 f 和

$f0$: `f(x){ S }` $\Longrightarrow$

```
f(x){
  f0(y){ S[x ↦ y, f ↦ f0] }
  f0(x);
}
```

所有外部函数调用都指向函数 f，而所有内部调用都指向函数 $f0$，外加一个从函数 f 对函数 $f0$ 的调用。

将该变换作用于前述例子，我们将得到

```
sum(k, x){
  sum0(k, x){
    if(x == 0) k(0);
    else{
      f(a) { k (a + x); }
      sum0(f, x-1); }
  }
  sum0(k, x);
}
g(k, y){
  sum0(k, x){
    if(x == 0) k(0);
    else{
      f(a) { k (a + x); }
      sum0(f, x-1); }
  }
  sum0(k, y);
}
```

函数 `g` 对函数 `sum` 的调用被内联后，将只调用函数 `sum` 内部的函数 `sum0`，而不再包含对函数 `sum` 的调用。

最后，我们还要指出的是，内联展开经常可以级联的方式进行，即把某个函数 `f` 内联展开到函数 `g` 内部后，函数 `g` 可继续在其调用点内联展开。为此，我们可以建立函数调用图，然后从图的叶子节点开始，向根节点的方向进行内联展开。

尽管内联展开能够降低函数调用带来的开销，但是由于将函数调用展开成了函数体，因此，内联展开往往会增加代码的规模。为此，可引入启发式策略来控制内联展开的进行，例如：

1) 仅对热点函数进行展开：热点函数的判定既可以基于静态的方式（如函数的嵌套深度）；也可以基于动态的方式（如基于反馈的优化）；

2) 仅对函数体大小不超过某个阈值的函数进行展开：这样，因内联展开而造成的代码增量可控；
3) 仅对调用次数不超过某个阈值的函数进行展开。

B. 尾递归优化

函数式程序经常使用（递归）函数来表达计算和控制流结构，而命令式语言更常用循环结构。因此，函数式语言的编译器需要对函数调用/返回进行特定优化，以期达到和循环相近的执行效率。例如，对于上述讨论的内联展开后得到的程序：

```
g(k, y){
  sum0(k, x){
    if(x == 0) k(0);
    else{
      f(a) { k (a + x); }
      sum0(f, x-1); }
  }
  sum0(k, y);
}
```

如果不加优化的把对函数 `sum0` 的调用编译为函数调用，则每次调用都会伴随调用栈创建和销毁，带来可观的运行时间开销，并且调用栈会随着调用的进行不断扩展。

尾递归优化（Tail recursion optimization）通过把尾（递归）函数调用优化为循环，从而有效降低函数调用执行时间和减小执行所需空间。直观上，*尾调用*（Tail call）指的是一个函数调用发生在某个计算过程的“最后一步”；注意，最后一步指的是计算过程，而未必是计算静态语法形式的最后一个部分。例如，考虑以下示例：

1) `h(a){ f(x); g(y); }`
2) `h(a){ f(g(y)); }`
3) `h(a){ if(f(x)) then g(y) else k(z); }`
4) `h(a){ f(x) + g(y); }`

在第一个函数定义中，调用 `g(y)` 是一个尾调用；而在第二个函数定义中，函数调用 `g(y)` 并不是尾调用，因为它的结果还要作为参数，调用函数 `f`；在第三个函数定义中，`f(x)` 不是尾调用，因为它的结果要用在条件判断中，而函数 `g(y)` 和 `k(z)` 都是尾调用（尽管调用 `g(y)` 并未出现在语法结构的尾部，但它仍然是整个计算的最后一步）；在第四个函数定义中，函数调用 `f(x)` 和 `g(y)` 的值要用于完成加法的计算，因此它们都不是尾调用。特别需要指出的是，本章讨论的 λ^A 系统中，所有的调用，都是尾调用，我们把对该结论的验证作为练习留给读者完成。

尾（递归）调用可以被编译为跳转，因而可以更加高效的实现函数调用。例如，考虑上述的第一个例子：

```
h(a){ f(x); g(y); }
```

由于函数调用 `g(y)` 是一个尾调用，其返回值也将是函数 `h` 的返回值。因此，我们可以在对函数 `g` 调用前，恢复函数 `h` 的执行现场，并跳转到函数 `g` 执行；这样，如果函数 `g` 执行返回，将直接返回 `h` 的调用者。

具体地，尾调用优化的具体步骤包括（“当前函数”指的是尾调用所处的函数，如上述的函数 `h`）：

1) 恢复当前函数的执行环境，包括恢复被调用者保存寄存器等；
2) 弹掉当前函数的调用栈帧，即让调用栈恢复到当前函数刚被调用时的状态；
3) 放置尾调用函数的参数，参数可能位于寄存器、栈之一，或两者兼有；
4) “跳转”到尾调用函数执行。

步骤 1 和 2 的目的都是让执行环境恢复到函数 `h` 刚要开始执行的状态，即只放置了 `h` 的返回地址；而步骤 3 是放置待调用的尾函数的参数，根据具体目标指令集体系结构调用规范的不同，这些参数可能位于参数寄存器或调用栈上，也可能同时位于这两者；最后，步骤 4 跳转到待调用的尾巴函数执行。

我们接下来分析每个步骤的可能需要实际执行的工作。首先，如果寄存器分配优先使用调用者保存寄存器，则有可能不会分配被调用者保存寄存器，因此步骤 1 实际上不涉及任何操作。其次，在拥有较多寄存器的指令集体系结构上（如本文中使用的拥有 32 个寄存器的 RISC-V），溢出可能不会发生，因此栈帧不需要进行任何调整，即步骤 2 不涉及任何操作。第三，如果在寄存器分配阶段，将尾调用的函数参数和参数寄存器进行接合，则步骤三不涉及任何操作。最后，跳转到目标尾函数执行，其时间开销和循环中的跳转时间开销相同。

利用尾递归优化，上述求和函数 `sum`（经过了闭包变换），可被优化为如下只包括显式跳转 `goto` 的函数：

```
sum: //
  if(x == 0)
    goto k;
```

```
  else
    f = newClosure(...);
    goto sum
```

不难验证：该函数和命令式编译器中生成的循环结构非常类似。另外，上述函数式风格的程序需要在堆上显式分配闭包，而命令式风格的程序需要显式的分配栈帧；尽管一般栈分配效率可能更高，但仔细调优堆分配器和垃圾收集器，堆分配也可以达到较高的执行效率。我们把对代码结构和执行效率等方面的细致比较，作为练习留给读者。

VI. 静态单赋值形式与函数式编程

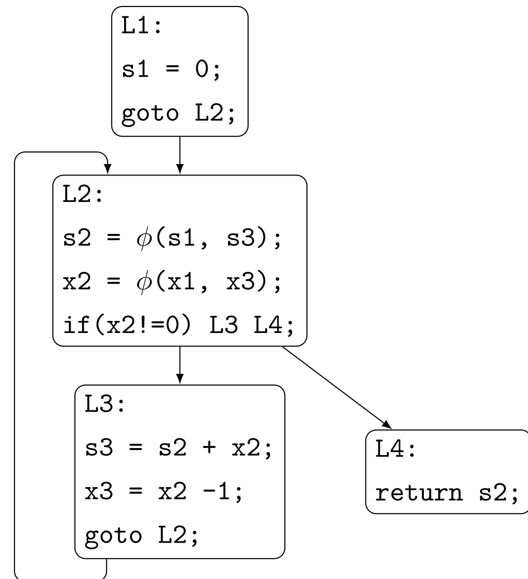
命令式语言编译器经常使用静态单赋值形式作为其中间表示，其用基本块表示计算且用有向边表示跳转关系；函数式语言的编译器经常使用延续传递风格作为其中间表示，其用延续函数表示基本计算，且用函数调用表示跳转关系。实际上，在很大程度上，尽管静态单赋值形式和延续传递风格从语法形式上看起来很不一样，但两者其实表达了同一种中间表示的两种不同形式。本小节，我们讨论两种重要中间表示的关系；主要内容包括基本块与延续、静态单赋值形式和延续传递风格的相互转换。

A. 基本块与延续

静态单赋值形式的控制流图的基本块中可能包括 ϕ 函数，代表对从不同控制流得到的值的选择。例如，命令式风格的求和函数 sum：

```
sum(x){
  s = 0;
  while(x != 0){
    s = s + x;
    x = x - 1;
  }
  return s;
}
```

被转换为静态单赋值形式后，得到控制流图：



我们可以将每个基本块看成一个延续函数，其参数个数等于基本块中 ϕ 函数的个数；特别的，没有 ϕ 函数的基本块将不接受参数。同时，基本块间的跳转关系，将转换为对基本块所对应函数的调用关系；特别的，如果基本块包括 ϕ 函数，则调用时，将传递 ϕ 函数中相应的参数。

按上述思想，上述静态单赋值程序可被转换为：

```
L1(){
  L2(s2, x2){
    L3(){
      s3 = s2 + x2
      x3 = x2 - 1
      L2(s3, x3)
    }
    L4(){
      k(s2)
    }
    if(x != 0) L3()
    else L4()
  }
  s1 = 0
  L2(s1, x1)
}
```

其中的基本块及其跳转关系都已被相应转换完成，共计生成了四个延续 L1 到 L4。以基本块 $L2$ 为例，它被转换为具有两个参数 $s2$ 和 $x2$ 的延续 L2(s2, x2){...}；而从基本块 $L1$ 和 $L3$ 到基本块 $L2$ 的跳转，分别被转换为两个函数调用 L2(s1, x1) 和 L2(s3, x3)。

需要特别注意，生成的延续必须满足特定的嵌套关

系，才能保证变量位于在合法的词法作用域内。例如，在上述示例中，延续 `L2` 嵌套在延续 `L1` 中，而延续 `L3` 和 `L4` 都嵌套在延续 `L2` 中；这种嵌套顺序能够使得延续访问其词法作用域中的变量，例如，延续 `L3` 可以访问延续 `L2` 中的变量 `s2` 和 `x2`。

要生成合法的延续嵌套结构，编译器需要利用必经节点树，即从树的根节点开始，按照对树的层序遍历的顺序，为每个基本块生成延续，每个基本块 B 的孩子节点 C_1、...、C_n 生成的延续，将直接嵌套在基本块 B 生成的延续中。请读者自行验证，上述示例程序中的延续嵌套结构，满足必经节点树的结构。

B. 静态单赋值形式转换为延续传递风格

可以用严格的规则，描述从静态单赋值形式到延续传递风格的转换规则。为此，我们首先给出一个简化的静态单赋值形式的语法规则：

ϕ 函数	i	$::=$	$x = \phi(y_1, \ldots, y_n)$
语句	s	$::=$	$x = n \mid x = y \mid x = y \oplus z \mid x = f(y)$
转移	t	$::=$	$\texttt{goto}\ L \mid \texttt{if}(\ldots)\ L_1\ L_2 \mid \texttt{return}\ x$
基本块	b	$::=$	$L : \vec{i}\ \vec{s}\ t$
函数	f	$::=$	$x(y)\{\vec{b}\}$
程序	p	$::=$	$\vec{f}$

每个 ϕ 函数接受 y_i，$1 \leq i \leq n$ 作为参数；语句 s 包括常量赋值、变量赋值、二元操作、函数调用等语法形式；转移 t 代表控制跳转，包括无条件跳转、条件跳转、返回等；基本块 b 以标号 L 开头，后跟 ϕ 函数列表 $\vec{i}$、语句列表 $\vec{s}$，并以单独的转移 t 结尾；函数 f 包括一个函数名 x、一个参数 y（注意，包含多个参数的函数与此类似）、以及基本块列表 b 构成的函数体；程序 p 包括函数列表 $\vec{f}$。和前面的约定类似，我们经常用显式的展开形式 f_1，...，f_n 来表示列表 $\vec{f}$。

我们以下标形式的编译函数簇 $[\![\cdot]\!]$，分别表示将相应的语法结构，从静态单赋值形式翻译到延续传递风格中的对应形式。具体的，对于程序 $p = f_1 \ldots f_n$，编译规则

$$[\![f_1 \ldots f_n]\!]_p = [\![f_1]\!]_f \ldots [\![f_n]\!]_f$$

分别编译程序 p 中的每个函数 f_i，$1 \leq i \leq n$，从而生成延续传递风格中的一组函数列表。

而对一个函数 $x(y)\{\vec{b}\}$，编译规则

$$[\![x(y)\{\vec{b}\}]\!]_f = x(k, y)\{[\![b_1]\!]_b \ldots [\![b_n]\!]_b\}$$

给函数 x 增加了一个延续参数 k，并递归编译函数体中的所有基本块 b_i，$1 \leq i \leq n$，分别得到对应延续 $[\![b_i]\!]_b$，$1 \leq i \leq n$。编译得到所有延续函数必须满足特定的嵌套顺序，我们将在下面的算法 3中进行讨论。另外需要注意，由于接下来的所有编译规则都需要函数的延续参数 k，我们令其作为接下来规则的隐式参数，因而可将延续参数 k 从编译规则中略去。

对于基本块 b，编译规则

$$
\begin{aligned}
&[\![L : x_1 = \phi(\ldots); \ldots; x_n = \phi(\ldots)\ \vec{s}\ t]\!]_b \\
&= L(x_1, \ldots, x_n)\{ \\
&\qquad [\![\vec{s}]\!]_s \\
&\qquad [\![t]\!]_t \\
&\quad \}
\end{aligned}
$$

对具有 ϕ 函数列表 $x_1 = \phi(\ldots); \ldots; x_n = \phi(\ldots)$ 的基本块 b，生成一个具有 n 个参数 $x_1, \ldots, x_n$ 的函数 L；同时，函数体包括基本块 b 中语句列表 $\vec{s}$ 的编译结果 $[\![\vec{s}]\!]_s$、以及最后转移语句 t 的编译结果 $[\![t]\!]_t$ 两部分。

对语句列表 $\vec{s}$，编译规则

$$
\begin{aligned}
[\![x = n; \vec{s}; t]\!]_s &= x = n; [\![\vec{s}; t]\!]_s \\
[\![x = y; \vec{s}; t]\!]_s &= x = y; [\![\vec{s}; t]\!]_s \\
[\![x = y \oplus z; \vec{s}; t]\!]_s &= x = y \oplus z; [\![\vec{s}; t]\!]_s \\
[\![x = f(y); \vec{s}; t]\!]_s &= h(a)\{\ x = a; [\![\vec{s}; t]\!]_s\ \} \\
&\qquad f(h, y) \\
[\![\cdot; t]\!]_s &= [\![t]\!]_t
\end{aligned}
$$

对列表中第一条语句的语法形式进行归纳：对基本的常量赋值、变量赋值、二元运算等语句，规则将其直接编译为延续传递风格中对应的语法形式；而对于函数调用 $x = f(y)$，规则新生成一个延续 h，并将其作为额外的第一参数，传递给函数 f，从而构成新的调用 $f(h, y)$；注意，生成的该新的调用成为最后一个语句。最后，当所有语句 s 处理完后，规则 $[\![\cdot; t]\!]_s$ 将调用对转移 t 的编译规则 $[\![t]\!]_t$。

对于转移 t，规则

$$
\begin{aligned}
[\![\texttt{goto}\ L]\!]_t &= L(x_1, \ldots, x_n) \\
[\![\texttt{if}(\ldots)\ L_1\ L_2]\!]_t &= \texttt{if}(\ldots)\ L_1(\ldots)\ L_2(\ldots) \\
[\![\texttt{return}\ x]\!]_t &= k(x)
\end{aligned}
$$

算法 3： 延续嵌套布局

输入： p：静态单赋值形式的程序

输出： p'：延续传递风格的程序

1: **procedure** NESTLAYOUT(p)
2: **for** 每个函数定义 $x(y)\ \{B_1 \dots B_n\}$ **do**
3: ▷ 构造函数 x 的必经节点树 t
4: $t = domTree(x)$
5: **for** 每个树节点 $u \in T$（按层序遍历顺序） **do**
6: **for** u 在树 t 中的每个直接孩子节点 v **do**
7: 将 $[\![v]\!]_b$ 直接嵌套在 $[\![u]\!]_b$ 中
8: **return** p'

根据对 t 的语法形式进行归纳：对无条件转移 `goto`，规则生成一个延续调用 $L(x_1, \dots, x_n)$，其中 x_i，$1 \le i \le n$ 是跳转的目标基本块 L 的 n 个 ϕ 函数的对应参数；特别的，如果该目标基本块 L 不包括 ϕ 函数，则 $n = 0$，该延续调用退化成无参形式 $L()$。对有条件转移 `if`，规则分别编译其两个跳转目标 L_1 和 L_2，并分别生成两个对应的函数 $L_1(\dots)$ 和 $L_2(\dots)$。最后，对于函数返回 `return`，规则将当前函数的延续 k，直接作用到返回值 x 上，形成延续调用 $k(x)$。

算法 3给出了对函数内基本块生成的延续进行布局的主要步骤。算法接受静态单赋值形式的程序 p 作为输入，将其转换为延续传递风格的程序 p'。首先，算法为待转换的函数 $x(y)\{B_1 \dots B_n\}$ 构建必经节点树 t，按照从叶子节点向根节点的层序遍历的顺序对树 t 进行遍历，对遍历到的节点 u，假设其直接孩子节点为 v_i，$1 \le i \le n$，则将孩子节点 v_i 编译得到的延续 $[\![v_i]\!]_b$ 直接嵌套在节点 u 编译得到的延续 $[\![u]\!]_b$ 中。

C. 延续传递风格转换为静态单赋值形式

延续传递风格的程序，同样可以转换为等价的静态单赋值形式。为此，我们同样引入以下标形式的编译函数簇 $[\![\cdot]\!]$，分别表示将相应的语法结构，从延续传递风格编译到静态单赋值形式中的对应形式的规则。

对程序 $p = f_1 \dots f_n$，编译规则

$$[\![f_1 \dots f_n]\!]_p = [\![f_1]\!]_f \dots [\![f_n]\!]_f$$

分别编译程序 p 中的每个函数 f_i，$1 \le i \le n$，从而生成静态单赋值形式中的一组函数列表 $[\![f_i]\!]_f$，$1 \le i \le n$。

而对一个函数 $x(k, y)\{e\}$，编译规则

$$[\![x(k, y)\{e\}]\!]_f = x(y)\{[\![e]\!]_e\}$$

移除了函数 x 的延续参数 k，并将编译函数体中表达式 e 的结果，作为新的函数体。

对表达式 e，编译规则 $[\![e]\!]$ 基于对表达式 e 语法形式的归纳，其编译为静态单赋值形式：

$$\begin{aligned}
[\![L(x_1, \dots, x_n)\{e\}]\!] &= L: \\
&\quad x_1 = \phi(\dots); \\
&\quad \dots; \\
&\quad x_n = \phi(\dots); \\
&\quad [\![e]\!] \\
[\![x = n; e]\!] &= x = n; [\![e]\!] \\
[\![x = y; e]\!] &= x = y; [\![e]\!] \\
[\![x = y \oplus z; e]\!] &= x = y \oplus z; [\![e]\!] \\
[\![h(a)\{x = a; e\} f(h, y)]\!] &= x = f(y); [\![e]\!] \\
[\![L(x_1, \dots, x_n)]\!] &= \texttt{goto}\ L \\
[\![\texttt{if}(\dots) L_1(\dots) L_2(\dots)]\!] &= \texttt{if}(\dots) L_1\ L_2 \\
[\![k(x)]\!] &= \texttt{return}\ x
\end{aligned}$$

对于延续定义 $L(x_1, \dots, x_n)\{e\}$，规则将其转换为以 n 个 ϕ 函数开头基本块 L，其中对延续体 e 的编译结果 $[\![e]\!]$，将作为基本块 L 的组成语句。需要注意，ϕ 函数的参数，需要在编译延续调用的时候确定。

对常量赋值 $x = n$、变量赋值 $x = y$、算术运算 $x = y \oplus z$ 的转换，可直接递归进行；而对局部延续 h，编译规则直接生成函数调用 $f(y)$。

对延续调用 $L(x_1, \dots, x_n)$，规则将生成显式的跳转 `goto`，其中参数 x_i，$1 \le i \le n$ 将放置在基本块 L 包括的 ϕ 函数的对应位置。类似的，有条件跳转 `if` 也将延续调用，编译为跳转。最后，对函数参数延续 k 的调用 $k(x)$，将被编译为显式的返回语句 `return` x。

VII. 习题

1。请给出具体的程序示例，说明基于平坦闭包表示，具有 n 层嵌套函数定义的程序，最多将消耗 $O(n^2)$ 的闭包存储空间。

2。对于链式闭包，请结合具体的程序示例，说明对于有 n 层嵌套函数定义的程序，其环境构建最坏运行时间复

杂度为 $O(1)$、环境的存储空间存储复杂度为 $O(n)$、自由变量的访问时间复杂度为 $O(n)$。

3。请给出基于链式闭包表示的闭包变换算法。

4。对递归求和函数 `sum` 和斐波那契函数 `fib`，请给出延续传递风格变换的具体过程。

5。请给出从语言 λ^L 到 λ^A 的完整编译规则。

6。请给出示例程序 $(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4\ 5$ 编译到 λ^A 的完整结果程序。

7。请修改从 λ^A 编译到 RISC-V 的规则，加入调用者/被调用者寄存器的使用规范。

8。请给出示例程序 $(\lambda x.\lambda y.\lambda z.(x+y+z))\ 3\ 4\ 5$ 编译到 RISC-V 的完整汇编程序。

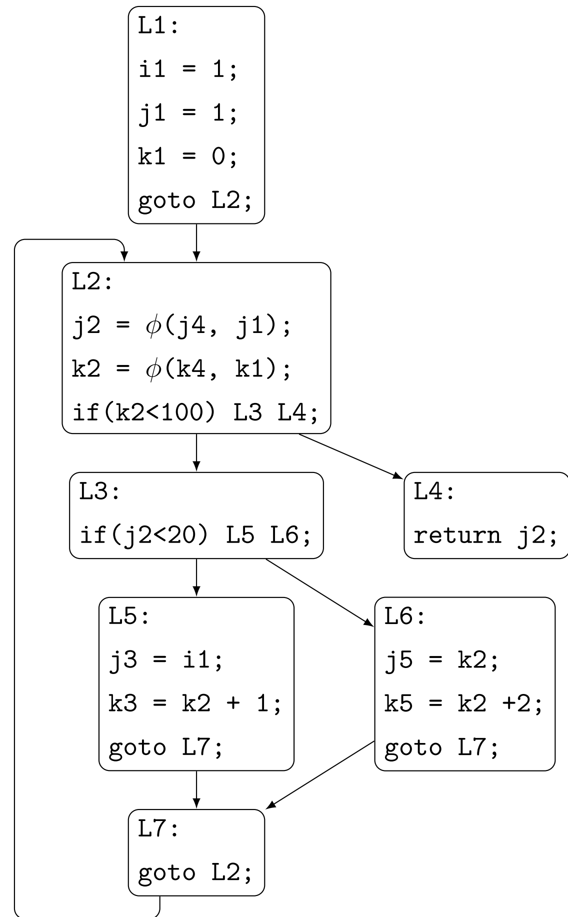
9。请验证 λ^A 所有的程序都包含尾调用形式。

10。对如下递归实现的求和程序

```
sum(x){
  if(x == 0)
    return 0;
  else return x + sum(x-1);
}
```

请给出其生成的目标代码，并与本章中讨论的经过延续传递风格变换和尾递归优化后，得到的目标代码结构比较，讨论其相似性以及执行效率间的关系。

11。请将如下静态单赋值形式的程序，转换为等价的延续传递风格的程序。



12。请将上述得到的延续传递风格的程序，重新转换为静态单赋值程序，并比较与转换前的静态单赋值程序间的异同。

参考文献

[1] Andrew W. Appel. 2007. Compiling with Continuations. Cambridge University Press, USA.

[2] Andrew Kennedy. 2007. Compiling with continuations, continued. In Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (ICFP '07). Association for Computing Machinery, New York, NY, USA, 177 – 190. https://doi.org/10.1145/1291151.1291179

[3] RISC-V specification. https://riscv.org/technical/specifications/

[4] Richard A. Kelsey. 1995. A correspondence between continuation passing style and static single assignment form. SIGPLAN Not. 30, 3 (March 1993), 13–22. https://doi.org/10.1145/202530.202532

[5] Andrew W. Appel. 1998. SSA is functional programming. SIGPLAN Not. 33, 4 (April 1998), 17 – 20. https://doi.org/10.1145/278283.278285