第九讲　函数式语言的编译

Compilation of Functional Languages

主要内容

- 函数式编程语言 micro-ML
- 中间表示：若干种正规形式
- 前端到中端：源程序到正规形式的转换
- 中端到后端：正规形式程序的闭包转换
- 中端上的优化

- 参考教材：Peter Sestoft. 2017. Programming Language Concepts. Springer Cham. DOI: 10.1007/978-3-319-60789-4

主要内容

- **函数式编程语言 micro-ML**
- 中间表示：若干种正规形式
- 前端到中端：源程序到正规形式的转换
- 中端到后端：正规形式程序的闭包转换
- 中端上的优化

表达式语言

- 让我们先从**表达式语言**开始

```
x - 11
```

```
(x - 11) * 2
```

x 是表达式中的自由变量(free variable)

```
if (x - 11) == 0 then y + 2 else y / 3
```

x 和 y 是表达式中的自由变量

```
let x = 7 in
  let y = 2 in
    if (x - 11) == 0 then y + 2 else y / 3
  end
end
```

let x = exp1 in exp2 end 是引入变量绑定的表达式，它首先计算 exp1 的值并绑定到 x 上，再计算并返回 exp2 的值

```
let x = 7 in
  let y = 2 in
    let y = let x = x - 1 in x + y end
    in x - y
    end
  end
end
```

let 表达式可以嵌套，并且可以引入重复的变量绑定，需要考虑变量的作用域问题

表达式语言的语法

- 本讲不考虑语法分析问题，直接讨论**抽象语法**

$$\mathit{Exp} ::= \mathsf{const}(n) \mid \mathsf{op}(\mathit{Exp}_1, \mathit{Exp}_2) \mid \mathsf{if}(\mathit{Exp}_1, \mathit{Exp}_2, \mathit{Exp}_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, \mathit{Exp}_1, \mathit{Exp}_2)$$

```
(x - 11) * 2
```

→ mul(sub(var("x"), const(11)), const(2))

```
if (x - 11) == 0 then y + 2 else y / 3
```

→ if(eq(sub(var("x"), const(11)), const(0)), add(var("y"), const(2)), div(var("y"), const(3)))

```
let x = 7 in
  let y = 2 in
    if (x - 11) == 0 then y + 2 else y / 3
  end
end
```

→ let("x", const(7), let("y", const(2), ⋯))

表达式语言的解释求值

- **如何构造一个属性文法对表达式进行求值?**
- 对表达式求值需要知道其中所有**自由变量**的值
- **环境(environment)**:维护自由变量到值的绑定的数据结构
  - 用符号 ρ 来指代环境
  - 用 $[]$ 表示空环境,即没有任何变量绑定
  - 用 $[var \mapsto val]\rho$ 表示在 ρ 上扩展 var 到 val 的绑定后得到的新环境
    - 允许有相同的变量绑定,后绑定的「覆盖」先绑定的
  - 用 $[var_1 \mapsto val_1, var_2 \mapsto val_2]\rho$ 表示 $[var_1 \mapsto val_1]([var_2 \mapsto val_2]\rho)$
  - 用 $[var_1 \mapsto val_1, var_2 \mapsto val_2, \cdots]$ 描述一个完整的环境

表达式语言的解释求值

$\rho = [\mathtt{x} \mapsto 20]$

```
(x - 11) * 2
```

求值结果为 18

$\rho = [\mathtt{y} \mapsto 2, \mathtt{x} \mapsto 11]$

```
if (x - 11) == 0 then y + 2 else y / 3
```

求值结果为 4

初始为空环境

$\rho = []$

```
let x = 7 in
  let y = 2 in
    let y = let x = x - 1 in x + y end
    in x - y
    end
  end
end
```

$\rho_1 = [\mathtt{x} \mapsto 7]$

$\rho_2 = [\mathtt{y} \mapsto 2, \mathtt{x} \mapsto 7]$

$\rho_3 = [\mathtt{x} \mapsto 6, \mathtt{y} \mapsto 2, \mathtt{x} \mapsto 7]$

$\rho_4 = [\mathtt{y} \mapsto 8, \mathtt{y} \mapsto 2, \mathtt{x} \mapsto 7]$

求值结果为 -1

表达式语言的解释求值

$Exp ::= \mathsf{const}(n) \mid \mathsf{op}(Exp_1, Exp_2) \mid \mathsf{if}(Exp_1, Exp_2, Exp_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, Exp_1, Exp_2)$

综合属性 val 记录求值结果

继承属性 ρ 记录求值环境

产生规则	属性计算规则
$E \to \mathsf{const}(n)$	$E.\mathtt{val} = n$
$E \to \mathsf{op}(E_1, E_2)$	$E_1.\rho = E.\rho \quad E_2.\rho = E.\rho$ $E.\mathtt{val} = f_{\mathsf{op}}(E_1.\mathtt{val}, E_2.\mathtt{val})$
$E \to \mathsf{if}(E_1, E_2, E_3)$	$E_1.\rho = E.\rho \quad E_2.\rho = E.\rho \quad E_3.\rho = E.\rho$ $E.\mathtt{val} = E_1.\mathtt{val}\ ?\ E_2.\mathtt{val}\ :\ E_3.\mathtt{val}$
$E \to \mathsf{var}(x)$	$E.\mathtt{val} = E.\rho(x)$
$E \to \mathsf{let}(x, E_1, E_2)$	$E_1.\rho = E.\rho \quad \boxed{E_2.\rho = [x \mapsto E_1.\mathtt{val}]E.\rho}$ $E.\mathtt{val} = E_2.\mathtt{val}$

表达式语言的解释求值

- 为了方便后面讨论(比如递归)，使用**逻辑系统**表述求值规则
- 用 $\rho \vdash E \Rightarrow v$ 表示**三元关系** (ρ, E, v)，表示「在环境 ρ 下对表达式 E 进行求值的结果是 v」

横线上方表示「前提」

$$\frac{}{\rho \vdash \mathsf{const}(n) \Rightarrow n} \qquad \frac{\rho \vdash E_1 \Rightarrow v_1 \quad \rho \vdash E_2 \Rightarrow v_2 \quad v = f_{\mathsf{op}}(v_1, v_2)}{\rho \vdash \mathsf{op}(E_1, E_2) \Rightarrow v}$$

横线下方表示「结论」

$$\frac{\rho \vdash E_1 \Rightarrow \mathsf{true} \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \mathsf{if}(E_1, E_2, E_3) \Rightarrow v} \qquad \frac{\rho \vdash E_1 \Rightarrow \mathsf{false} \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \mathsf{if}(E_1, E_2, E_3) \Rightarrow v} \qquad \frac{\rho(x) = v}{\rho \vdash \mathsf{var}(x) \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \boxed{[x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}}{\rho \vdash \mathsf{let}(x, E_1, E_2) \Rightarrow v}$$

一阶函数

- 在表达式语言的基础上加入**一阶函数**，即函数的参数和返回值必须是标量值(比如布尔值、整型值)

```
let f x = x - 11 in f (f 77) end
```

定义了一个名为 f 的函数

```
let f x =
  let g y = x + y in g (2 * x) end
in
  f 7
end
```

定义了一个名为 f 的函数，其内部嵌套定义了一个名为 g 的函数，而 g 的函数体使用了 f 函数的参数

```
let f x =
  if x == 0 then 1 else 2 * f (x - 1)
in
  f y
end
```

定义了一个名为 f 的递归函数，整个表达式计算的返回值为 2^y

一阶函数的解释求值

$$Exp ::= \cdots \mid \mathsf{letfun}(f, x, Exp_1, Exp_2) \mid \mathsf{call}(f, Exp_1)$$

- 需要在环境中记录函数名到函数定义的映射
  - 对于 $\mathsf{letfun}(f, x, E_1, E_2)$，记录 $\mathsf{fun}(f, x, E_1)$ 是否足够？

$\rho = [\mathsf{f} \mapsto \mathsf{fun}(\mathsf{f}, \mathsf{x}, \mathsf{sub}(\mathsf{var}(\text{"x"}), \mathsf{const}(11)))]$

```
let f x = x - 11 in f (f 77) end
```

求值结果为 55

$\rho = [\mathsf{g} \mapsto \mathsf{fun}(\mathsf{g}, \mathsf{y}, \mathsf{add}(\mathsf{var}(\text{"x"}), \mathsf{var}(\text{"y"})), \mathsf{x} \mapsto 7]$

```
let f x =
  let g y = x + y in g (2 * x) end
in
  f 7
end
```

对 g 的函数调用求值时，需要知道 g 被加入环境时的 x 的值

为什么需要这个限制？

一阶函数的解释求值

```
let f x =
  let g y = x + y in
    let x = 42 in g (2 * x) end
  end
in
  f 7
end
```

$\rho = [\mathsf{x} \mapsto 42, \mathsf{g} \mapsto \mathsf{fun}(\mathsf{g}, \mathsf{y}, \mathsf{add}(\mathsf{var}(\text{"x"}), \mathsf{var}(\text{"y"})), \mathsf{x} \mapsto 7]$

在本讲中，我们使用「静态」作用域
（也称为「词法」作用域）

求值结果为 91

$$\frac{[f \mapsto \mathsf{fun}(f, x, E_1)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \mathsf{letfun}(f, x, E_1, E_2) \Rightarrow v}$$

这样做是否正确？

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \boxed{\rho = [\cdots][f \mapsto \mathsf{fun}(f', x', E_1')]\rho'} \qquad [x' \mapsto v_1]\rho' \vdash E_1' \Rightarrow v}{\rho \vdash \mathsf{call}(f, E_1) \Rightarrow v}$$

一阶函数的解释求值

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho = [\cdots][f \mapsto \mathsf{fun}(f', x', E_1')]\rho' \qquad \boxed{[x' \mapsto v_1]\rho' \vdash E_1' \Rightarrow v}}{\rho \vdash \mathsf{call}(f, E_1) \Rightarrow v}$$

$\rho_2 = [\mathtt{x} \mapsto 2]$

环境中没有 f 的定义

```
let f x =
  if x == 0 then 1 else 2 * f (x - 1)
in
  f 2
end
```

$\rho_1 = [\mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

为了支持递归调用，在环境中继续保留被调用函数的定义

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad \rho = [\cdots][f \mapsto \mathsf{fun}(f', x', E_1')]\rho' \qquad \boxed{[x' \mapsto v_1, f' \mapsto \cdots]\rho' \vdash E_1' \Rightarrow v}}{\rho \vdash \mathsf{call}(f, E_1) \Rightarrow v}$$

$\rho_2 = [\mathtt{x} \mapsto 2, \mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

$\rho_3 = [\mathtt{x} \mapsto 1, \mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

$\rho_4 = [\mathtt{x} \mapsto 0, \mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

```
let f x =
  if x == 0 then 1 else 2 * f (x - 1)
in
  f 2
end
```

求值结果为 4

高阶函数

- 函数的参数和返回值也可以是函数

```
let f x =
  let g y = x + y in g end
in
  (f 2) 3
end
```

返回一个函数，可以认为 f 是一个双参数函数(也称为「柯里化」)

第一次函数调用 f 2 返回一个「加二」函数，第二次函数调用计算结果为 5

```
let f x =
  let g y = x + y in g end
in
  let h t = t 3 in
    h (f 2)
  end
end
```

参数 t 是一个函数，它接受整型值作为参数

函数调用 f 2 返回一个「加二」函数，并作为参数传递给 h 函数

高阶函数的解释求值

- **需要在环境中记录函数名到函数定义的映射**
  - 对于 letfun(f, x, E_1, E_2)，记录 fun(f, x, E_1) 是否足够？

```
let f x =
  let g y = x + y in g end
in
  (f 2) 3
end
```

$\rho = [\mathtt{x} \mapsto 2, \mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

$\rho = [\mathtt{f} \mapsto \mathsf{fun}(\mathtt{f}, \mathtt{x}, \cdots)]$

fun(g, y, add(var("x"), var("y")))

去哪里找 x 的值？

虽然仍然是静态作用域，但是高阶函数可能使得参数 x 的「生命周期」超过函数 f 本身

- **闭包(closure)：在记录函数定义时，记录「当时」的环境**

$$\frac{[f \mapsto \boxed{\mathsf{clos}(f, x, E_1, \rho)}]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \mathsf{letfun}(f, x, E_1, E_2) \Rightarrow v}$$

高阶函数的解释求值

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \boxed{\rho(f) = \mathsf{clos}(f', x', E_1', \rho')} \quad [x' \mapsto v_1, f' \mapsto \cdots]\rho' \vdash E_1' \Rightarrow v}{\rho \vdash \mathsf{call}(f, E_1) \Rightarrow v}$$

```
let f x =
  let g y = x + y in g end
in
  (f 2) 3
end
```

$\rho_2 = [\mathtt{x} \mapsto 2, \mathtt{f} \mapsto \mathsf{clos}(\mathtt{f}, \mathtt{x}, \cdots, [])]$

$\mathsf{clos}(\mathtt{g}, \mathtt{y}, \mathsf{add}(\mathsf{var}(\texttt{"x"}), \mathsf{var}(\texttt{"y"})), \rho_2)$

$\rho_1 = [\mathtt{f} \mapsto \mathsf{clos}(\mathtt{f}, \mathtt{x}, \cdots, [])]$

$\mathsf{clos}(\mathtt{g}, \mathtt{y}, \mathsf{add}(\mathsf{var}(\texttt{"x"}), \mathsf{var}(\texttt{"y"})), \rho_2)$

$\rho_3 = [\mathtt{y} \mapsto 3, \mathtt{g} \mapsto \cdots]\rho_2$

- 函数调用的函数 f 也可以是任意的表达式了：

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad \boxed{\rho \vdash E_2 \Rightarrow \mathsf{clos}(f', x', E_1', \rho')} \quad [x' \mapsto v_1, f' \mapsto \cdots]\rho' \vdash E_1' \Rightarrow v}{\rho \vdash \mathsf{call}(E_2, E_1) \Rightarrow v}$$

高阶函数示例

- 下面这段代码在做什么计算？

$\rho_4 = [\mathtt{x} \mapsto \mathtt{0}, \mathtt{maker} \mapsto \mathsf{clos}(\mathtt{makefac}, \mathtt{maker}, \cdots, [])]$

$\rho_3 = [\mathtt{x} \mapsto \mathtt{1}, \mathtt{maker} \mapsto \mathsf{clos}(\mathtt{makefac}, \mathtt{maker}, \cdots, [])]$

$\rho_2 = [\mathtt{x} \mapsto \mathtt{2}, \mathtt{maker} \mapsto \mathsf{clos}(\mathtt{makefac}, \mathtt{maker}, \cdots, [])]$

$\rho_1 = [\mathtt{x} \mapsto \mathtt{3}, \mathtt{maker} \mapsto \mathsf{clos}(\mathtt{makefac}, \mathtt{maker}, \cdots, [])]$

```
let makefac maker =
  let f x = if x == 0 then 1 else x * ((maker maker) (x - 1))
  in f end
in
  let fac = makefac makefac
  in fac 3 end
end
```

$\mathsf{clos}(\mathtt{f}, \mathtt{x}, \cdots, [\mathtt{maker} \mapsto \mathsf{clos}(\mathtt{makefac}, \mathtt{maker}, \cdots, [])])$

求值结果为 6

- 即便不原生支持递归函数，高阶函数也能「模拟」递归

主要内容

- 函数式编程语言 micro-ML
- **中间表示：若干种正规形式**
- 前端到中端：源程序到正规形式的转换
- 中端到后端：正规形式程序的闭包转换
- 中端上的优化

函数式语言的中间表示

- 由于**高阶函数**的存在，之前讲的三地址代码等 IR 都不合适
  - 或者说，在转换为三地址代码前还需要一种 IR
- 回顾：三地址代码主要简化了嵌套表达式的结构
- 作为表达式语言，函数式语言的 IR 也注重简化嵌套结构
  - K-Normal Form：运算分量、分支条件不嵌套，`let` 可以随意嵌套
  - A-Normal Form：运算分量、分支条件、`let ... in` 中间部分不嵌套
  - Continuation-Passing Style：运算分量分支条件不嵌套，只有尾递归
  - **从上往下，嵌套结构越简单，转换算法越复杂**

K-Normal Form

$$E ::= \mathrm{const}(n) \mid \mathrm{op}(E_1, E_2) \mid \mathrm{if}(E_1, E_2, E_3) \mid \mathrm{var}(x) \mid \mathrm{let}(x, E_1, E_2) \mid \mathrm{letfun}(f, x, E_1, E_2) \mid \mathrm{call}(E_1, E_2)$$

- 运算分量、分支条件不嵌套，let 可以随意嵌套：

原子(atom)表达式

$$\boxed{A ::= \mathrm{const}(n) \mid \mathrm{var}(x)}$$

$$E ::= A \mid \mathrm{op}(A_1, A_2) \mid \mathrm{if}(A_1, E_2, E_3) \mid \mathrm{let}(x, E_1, E_2) \mid \mathrm{letfun}(f, x, E_1, E_2) \mid \mathrm{call}(A_1, A_2)$$

```
if (x - 11) == 0 then y + 2 else y / 3
```

$\Rightarrow$

```
let t1 = x - 11 in
  let t2 = t1 == 0 in
    if t2 then y + 2 else y / 3
  end
end
```

K-Normal Form

$$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$$

- 运算分量、分支条件不嵌套，let 可以随意嵌套：

原子(atom)表达式

$$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$$

$$E ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(A_1, A_2)$$

```
let f x =
  if x == 0 then 1 else 2 * f (x - 1)
in
  f (f 7)
end
```

→

```
let f x =
  let t1 = x == 0 in
    if t1 then 1 else
    let t2 = x - 1 in
      let t3 = f t2 in 2 * t3 end
    end
  end
in
  let t4 = f 7 in
    f t4
  end
end
```

K-Normal Form

$E ::= \text{const}(n) \mid \text{op}(E_1, E_2) \mid \text{if}(E_1, E_2, E_3) \mid \text{var}(x) \mid \text{let}(x, E_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \mid \text{call}(E_1, E_2)$

- 运算分量、分支条件不嵌套，let 可以随意嵌套：

原子(atom)表达式

$\boxed{A ::= \text{const}(n) \mid \text{var}(x)}$

$E ::= A \mid \text{op}(A_1, A_2) \mid \text{if}(A_1, E_2, E_3) \mid \text{let}(x, E_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \mid \text{call}(A_1, A_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

→

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

A-Normal Form

$$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$$

- 运算分量、分支条件、let ... in 中间部分不嵌套：

$$\begin{aligned} A &::= \mathsf{const}(n) \mid \mathsf{var}(x) \\ \boxed{C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2)} \\ E &::= C \mid \mathsf{if}(A_1, E_2, E_3) \mid \boxed{\mathsf{let}(x, C_1, E_2)} \mid \mathsf{letfun}(f, x, E_1, E_2) \end{aligned}$$

「复杂」表达式

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

→

```
let x = 7 in
  let y = 2 in
    let w = x - 1 in
      let z = w + y in
        x - z
      end
    end
  end
end
```

A-Normal Form

$$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$$

- 运算分量、分支条件、let ... in 中间部分不嵌套:

$$\begin{aligned}
A &::= \mathsf{const}(n) \mid \mathsf{var}(x) \\
C &::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \\
E &::= C \mid \mathsf{if}(A_1, E_2, E_3) \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)
\end{aligned}$$

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

分支表达式后面的代码会重复生成,可能造成代码大小指数增长

→

```
let f x =
  let t1 = x == 0 in
    if t1 then
    let y = 1 in let t2 = y + 2 in t2 / 5 end end
    else
    let t3 = x - 1 in
      let t4 = f t3 in
        let y = x * t4 in
          let t5 = y + 2 in t5 / 5 end end
        end end
  end
in f 7 end
```

A-Normal Form

$$E ::= \text{const}(n) \mid \text{op}(E_1, E_2) \mid \text{if}(E_1, E_2, E_3) \mid \text{var}(x) \mid \text{let}(x, E_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \mid \text{call}(E_1, E_2)$$

- 运算分量、分支条件、let ... in 中间部分不嵌套：

$$\begin{aligned} A &::= \text{const}(n) \mid \text{var}(x) \\ C &::= A \mid \text{op}(A_1, A_2) \mid \text{call}(A_1, A_2) \mid \boxed{\text{if}(A_1, E_2, E_3)} \\ E &::= C \mid \text{let}(x, C_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \end{aligned}$$

允许 if 出现在 let ... in 中间部分

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in
        let t3 = f t3 in x * t3 end
      end
    in
      let t4 = y + 2 in t4 / 5 end
    end
  end
in f 7 end
```

Continuation-Passing Style

- 「continuation」的概念比较难解释，本讲中考虑如下约束：
- **任何的函数调用都不会(或者说不需要)返回其调用者**
- 如何实现这个效果？

每个函数接受一个回调(callback)
来模拟函数返回

```
function id(x) {
  return x ;
}
```

```
function id(x, ret) {
  ret(x);
}
```

```
function fac(n) {
  if (n == 0)
    return 1;
  else
    return n * fac(n - 1);
}
```

构造一个新的回调

```
function fac(n, ret) {
  if (n == 0)
    ret(1);
  else
    fac(n - 1, function (t1) { ret(n * t1) });
}
```

Continuation-Passing Style

```
function fac(n) {
  return tail_fac(n, 1);
}

function tail_fac(n, a) {
  if (n == 0)
    return a;
  else
    return tail_fac(n - 1, n * a);
}
```

尾递归实现的阶乘

```
function fac(n, ret) {
  tail_fac(n, 1, ret);
}

function tail_fac(n, a, ret) {
  if (n == 0)
    ret(a);
  else
    tail_fac(n - 1, n * a, ret);
}
```

尾递归转换后不引入新回调

```
function fib(n) {
  if (n < 2)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```

```
function fib(n, ret) {
  if (n < 2)
    ret(1);
  else
    fib(n - 1,
      function (t1) {
        fib(n - 2, function (t2) { ret(t1 + t2) })
      });
}
```

第一处递归的回调

第二处递归的回调

CPS 作为一种编程方式

- 这种编程方式称为 Continuation-Passing Style (CPS)
- 在异步编程中其实挺常见的，通过回调来实现**非阻塞**计算
  - fetch(url) 同步调用，会阻塞当前计算直到网络访问返回
  - fetch(url, ret) 异步调用，不阻塞计算，返回后由 ret 回调处理
- 例如，考虑组合数计算函数，其中的阶乘是服务器远程计算：

```
function comb(n, k) {
  return fac(n) / (fac(k) * fac(n - k));
}

function fac(n) {
  var res = fetch("./fact/" + n);
  return eval(res);
}
```

→

```
function comb(n, k, ret) {
  fac(n, function (fn) {
    fac(k, function (fk) {
      fac(n - k, function (fnk) {
        ret(fn / (fk * fnk))
      })
    })
  });
}

function fac(n, ret) {
  fetch("./fact/" + n, function (res) { ret(eval(res)) });
}
```

CPS 作为一种编程方式

- 如果用 CPS 风格写程序，需要改变**异常处理**等控制流的实现

```
function fac(n) {
  if (n < 0)
    throw "n < 0";
  else if (n == 0)
    return 1;
  else
  return n * fac(n - 1);
}

function total_fac(n) {
  try {
    return fac(n);
  } catch (ex) {
    return false;
  }
}
```

可以有多个回调函数，这里的 thro 回调用于进行异常处理

```
function fac(n, ret, thro) {
  if (n < 0)
    thro("n < 0");
  else if (n == 0)
    ret(1);
  else
    fac(n - 1, function (t1) { ret(n * t1) }, thro);
}

function total_fac(n, ret) {
  fac(n, ret, function (ex) { ret(false) });
}
```

进行异常处理的回调

CPS 作为一种中间表示

$$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$$

- 运算分量、分支条件不嵌套，只有尾递归，**每个函数加上回调参数**：

$$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$$
$$C ::= A \mid \mathsf{op}(A_1, A_2)$$
$$E ::= C \mid \mathsf{if}(A_1, E_2, E_3) \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, \overrightarrow{x_1, \boxed{\cdots, x_n}}, E_1, E_2) \mid \mathsf{call}(A_0, \overrightarrow{A_1, \boxed{\cdots, A_n}})$$

```
let f x =
  if x == 0 then 1 else 2 * f (x - 1)
in
  f 7
end
```

→

```
let f x ret1 =
  let t1 = x == 0 in
    if t1 then ret1(1)
    else
    let t2 = x - 1 in
      let ret2 t3 = let t4 = 2 * t3 in ret1(t4) end
      in f t2 ret2 end
    end end
in
  let ret3 t5 = t5 in f 7 ret3 end
end
```

CPS 作为一种中间表示

$E ::= \text{const}(n) \mid \text{op}(E_1, E_2) \mid \text{if}(E_1, E_2, E_3) \mid \text{var}(x) \mid \text{let}(x, E_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \mid \text{call}(E_1, E_2)$

- 运算分量、分支条件不嵌套，只有尾递归，**每个函数加上回调参数**：

$$A ::= \text{const}(n) \mid \text{var}(x)$$
$$C ::= A \mid \text{op}(A_1, A_2)$$
$$E ::= C \mid \text{if}(A_1, E_2, E_3) \mid \text{let}(x, C_1, E_2) \mid \text{letfun}(f, \overrightarrow{x_1, \cdots, x_n}, E_1, E_2) \mid \text{call}(A_0, \overrightarrow{A_1, \cdots, A_n})$$

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  ...
end
```

分支表达式后面的代码会重复生成，可能造成代码大小指数增长

→

```
let f x ret1 =
  let t1 = x == 0 in
    if t1 then
    let y = 1 in let t2 = y + 2 in let t3 = t2 / 5 in
      ret1(t3) end end end
    else
    let t4 = x - 1 in
      let ret2 t5 = let y = x * t5 in
        let t6 = y + 2 in let t7 = t6 / 5 in
          ret1(t7) end end end
      in f t4 ret2 end
    end end in ... end
```

CPS 作为一种中间表示

$$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$$

- 运算分量、分支条件不嵌套，只有尾递归，**每个函数加上回调参数**

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  ...
end
```

```
let f x ret1 =
  let t1 = x == 0 in
    let retif y =
      let t2 = y + 2 in let t3 = t2 / 5 in
        ret1(t3) end end
    in
      if t1 then retif(1)
      else
      let t4 = x - 1 in
        let ret2 t5 =
          let y = x * t5 in retif(y) end
        in f t4 ret2 end
      end
    end
  end
in ... end
```

为分支表达式的后续计算引入一个新的回调

CPS 与 SSA

- 尾递归函数调用就像是控制流图上的 goto 语句
- 对于一阶函数，可以改写为支持带参数 goto 的Koopa IR 形式

```
let f x ret1 =
  let t1 = x == 0 in
    let retif y =
      let t2 = y + 2 in let t3 = t2 / 5 in
        ret1(t3) end end
    in
      if t1 then retif(1)
      else
      let t4 = x - 1 in
        let ret2 t5 =
          let y = x * t5 in retif(y) end
        in f t4 ret2 end
      end
    end
  end
in ... end
```

→

```
f(x):
  t1 = x == 0
  if t1 then
    goto retif(1)
  else
    t4 = x - 1
    t5 = f(t4)
    y = x * t5
    goto retif(y)

retif(y):
  t2 = y + 2
  t3 = t2 / 5
  return t3
```

主要内容

- 函数式编程语言 micro-ML
- 中间表示：若干种正规形式
- **前端到中端：源程序到正规形式的转换**
- 中端到后端：正规形式程序的闭包转换
- 中端上的优化

转换为 K-Normal Form (KNF)

$E ::= \mathsf{const}(n) \mid \mathsf{op}(E_1, E_2) \mid \mathsf{if}(E_1, E_2, E_3) \mid \mathsf{var}(x) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(E_1, E_2)$

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$E ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3) \mid \mathsf{let}(x, E_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2) \mid \mathsf{call}(A_1, A_2)$

- 设计一个**属性文法**，把函数式程序转换为 KNF 形式

产生规则	属性计算规则
$E \to \mathsf{const}(n)$	$E.\mathsf{knf} = \mathsf{const}(n)$
$E \to \mathsf{op}(E_1, E_2)$	$E.\mathsf{knf} = \mathsf{let}(t_1, E_1.\mathsf{knf}, \mathsf{let}(t_2, E_2.\mathsf{knf}, \mathsf{op}(\mathsf{var}(t_1), \mathsf{var}(t_2))))$
$E \to \mathsf{if}(E_1, E_2, E_3)$	$E.\mathsf{knf} = \mathsf{let}(t_1, E_1.\mathsf{knf}, \mathsf{if}(\mathsf{var}(t_1), E_2.\mathsf{knf}, E_3.\mathsf{knf}))$
$E \to \mathsf{var}(x)$	$E.\mathsf{knf} = \mathsf{var}(x)$
$E \to \mathsf{let}(x, E_1, E_2)$	$E.\mathsf{knf} = \mathsf{let}(x, E_1.\mathsf{knf}, E_2.\mathsf{knf})$
$E \to \mathsf{letfun}(f, x, E_1, E_2)$	$E.\mathsf{knf} = \mathsf{letfun}(f, x, E_1.\mathsf{knf}, E_2.\mathsf{knf})$
$E \to \mathsf{call}(E_1, E_2)$	$E.\mathsf{knf} = \mathsf{let}(t_1, E_1.\mathsf{knf}, \mathsf{let}(t_2, E_2.\mathsf{knf}, \mathsf{call}(\mathsf{var}(t_1), \mathsf{var}(t_2))))$

生成临时变量名

转换为 A-Normal Form (ANF)

$E ::= \text{const}(n) \mid \text{op}(E_1, E_2) \mid \text{if}(E_1, E_2, E_3) \mid \text{var}(x) \mid \text{let}(x, E_1, E_2) \mid \text{letfun}(f, x, E_1, E_2) \mid \text{call}(E_1, E_2)$

$$
\begin{aligned}
A &::= \text{const}(n) \mid \text{var}(x) \\
C &::= A \mid \text{op}(A_1, A_2) \mid \text{call}(A_1, A_2) \mid \text{if}(A_1, E_2, E_3) \\
E &::= C \mid \text{let}(x, C_1, E_2) \mid \text{letfun}(f, x, E_1, E_2)
\end{aligned}
$$

- 先不考虑函数，主要考虑如何将嵌套 let 表达式「序列化」：

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

处理 let z = ... in 中间的表达式时，要考虑上下文的情况，所以需要某种继承属性

```
let x = 7 in
  let y = 2 in
    let w = x - 1 in
      let z = w + y in
        x - z
      end
    end
  end
end
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后(可能已经生成了若干层 let)表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
k2 = fun (c1) { return let("x", c1, norm("let y ...", k1)); }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后(可能已经生成了若干层 let)表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后(可能已经生成了若干层 let)表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
k3 = fun (c1) { return let("y", c1, norm("let z ...", k1)); }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

```
k1 = fun (c) { return c; }
```

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
k4 = fun (c1) { return let("z", c1, norm("x - z", k1)); }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
k4 = fun (c1) { return let("z", c1, norm("x - z", k1)); }
k5 = fun (c1) { return let("w", c1, norm("w + y", k4)); }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \text{const}(n) \mid \text{var}(x)$

$C ::= A \mid \text{op}(A_1, A_2) \mid \text{call}(A_1, A_2) \mid \text{if}(A_1, E_2, E_3)$

$E ::= C \mid \text{let}(x, C_1, E_2) \mid \text{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let z =
      let w = x - 1 in w + y end
    in
      x - z
    end
  end
end
```

```
k1 = fun (c) { return c; }
k4 = fun (c1) { return let("z", c1, norm("x - z", k1)); }
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

```
k1 = fun (c) { return c; }
```

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let w = x - 1 in
      let z = w + y in
        x - z
      end
    end
  end
end
```

转换为 A-Normal Form (ANF)

```
function normalize(e, k) {
  if (e.kind == "let") {
    return normalize(e.e1, function (c1) {
      return exp("let", e.x, c1, normalize(e.e2, k));
    });
  }
  else if (e.kind == "var" || e.kind == "const")
    return k(e);
  else if (e.kind == "op") {
    return normalize(e.e1, function (c1) {
      var t1 = gentmp();
      return normalize(e.e2, function (c2) {
        var t2 = gentmp();
        return exp("let", t1, c1,
               exp("let", t2, c2,
               k(exp("op", exp("var", t1), exp("var", t2)))));
      });
    });
  } ...
}
```

继承属性是一个「函数」，接受一个表达式 C 表示 e 转换后（可能已经生成了若干层 let）表示其结果的表达式，返回在整个转换上下文中得到的表达式 E

当 c1 是原子表达式时，不用生成 let 绑定

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$

$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$

$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

```
let x = 7 in
  let y = 2 in
    let w = x - 1 in
      let z = w + y in
        x - z
      end
    end
  end
end
```

转换为 A-Normal Form (ANF)

$$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$$
$$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$$
$$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$$

```
else if (e.kind == "if") {
  return normalize(e.e1, function (c1) {
    var t1 = gentmp();
    return exp("let", t1, c1,
               k(exp("if", exp("var", t1),
                     normalize(e.e2, function (c) { return c; }),
                     normalize(e.e3, function (c) { return c; }))));
  })
}
else if (e.kind == "letfun") {
  return exp("let", e.f, e.x,
             normalize(e.e1, function (c) { return c; }),
             normalize(e.e2, k));
}
else if (e.kind == "call") {
  ... // similiar to the op case
}
```

if 是 C 型表达式，可以作为 k 的参数

如果在这里使用 k，可能导致代码大小指数增长

对函数体定义转换时不用考虑 k

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    if x == 0 then 1 else x * f (x - 1)
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
k2 = fun (c) { return c; }
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
k4 = fun (c1) { t1=gentmp(); return let(t1, c1, k3(if(t1, norm("1", id), norm("x * f(x-1)", id))); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k5 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k6 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k6 = fun (c) { return c; }
```

```
k7 = fun (c1) { return norm("f(x-1)", fun (c2) { t2=gentmp(); return let(t2, c2, k6(c1 * t2)); }); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k6 = fun (c) { return c; }
```

```
k8 = fun (c2) { t2=gentmp(); return let(t2, c2, k6("x" * t2)); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
k2 = fun (c) { return c; }
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
k6 = fun (c) { return c; }
k8 = fun (c2) { t2=gentmp(); return let(t2, c2, k6("x" * t2)); }
k9 = fun (c1) { return norm("x-1", fun (c2) { t2=gentmp(); return let(t2, c2, k8(c1 t2)); }); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else x * f (x - 1)
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
k2 = fun (c) { return c; }
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
k6 = fun (c) { return c; }
k8 = fun (c2) { t2=gentmp(); return let(t2, c2, k6("x" * t2)); }
k10 = fun (c2) { t2=gentmp(); return let(t2, c2, k8("f" t2)); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else
      let t2 = x - 1 in x * f t2 end
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k6 = fun (c) { return c; }
```

```
k8 = fun (c2) { t2=gentmp(); return let(t2, c2, k6("x" * t2)); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

```
k6 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let y =
    let t1 = x == 0 in
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    end
  in
    (y + 2) / 5
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k3 = fun (c1) { return let("y", c1, norm("(y + 2) / 5", k2)); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      (y + 2) / 5
    end
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      (y + 2) / 5
    end
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k11 = fun (c1) { t1=gentmp(); return norm("5", fun (c2) { return k2(t1 / c2); }); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      let t4 = y + 2 in t4 / 5 end
    end
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

```
k12 = fun (c2) { return k2(t1 / c2); }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      let t4 = y + 2 in t4 / 5 end
    end
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

```
k2 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      let t4 = y + 2 in t4 / 5 end
    end
  end
in
  f 7
end
```

```
k1 = fun (c) { return c; }
```

转换为 A-Normal Form (ANF)

```
let f x =
  let t1 = x == 0 in
    let y =
      if t1 then 1 else
      let t2 = x - 1 in let t3 = f t2 in x * t3 end end
    in
      let t4 = y + 2 in t4 / 5 end
    end
  end
in
  f 7
end
```

- 练习：设计一个**属性文法**，把函数式程序转换为 ANF 形式
- **挑战：如何把函数式程序转换为 CPS 形式？**

主要内容

- 函数式编程语言 micro-ML
- 中间表示：若干种正规形式
- 前端到中端：源程序到正规形式的转换
- **中端到后端：正规形式程序的闭包转换**
- 中端上的优化

以 ANF 为中间表示

- 如何处理嵌套函数、高阶函数？

```
let f x =
  let g y = x + y in g end
in
  (f 2) 3
end
```

→

```
let f x =
  let g y = x + y in g end
in
  let t1 = f 2 in t1 3 end
end
```

```
let f x =
  let g y = x + y in g end
in
  let h t = t 3 in
    h (f 2)
  end
end
```

→

```
let f x =
  let g y = x + y in g end
in
  let h t = t 3 in
    let t1 = f 2 in h t1 end
  end
end
```

转换为显式构造闭包的形式

- 希望转换后所有的函数都是「最外层」函数
- 需要显式进行闭包的构造

「最外层」函数用 @ 标记

函数所需的闭包参数

显式构造闭包

```
let f x =
  let g y = x + y in g end
in
  (f 2) 3
end
```

→

```
let f x =
  let g y = x + y in g end
in
  let t1 = f 2 in t1 3 end
end
```

→

```
fun @f {} x =
  let g = clos @g {x} in g end
fun @g {x} y = x + y

let f = clos @f {} in
let t1 = f 2 in t1 3 end end
```

```
let f x =
  let g y = x + y in g end
in
  let h t = t 3 in
    h (f 2)
  end
end
```

→

```
let f x =
  let g y = x + y in g end
in
  let h t = t 3 in
    let t1 = f 2 in h t1 end
  end
end
```

→

```
fun @f {} x =
  let g = clos @g {x} in g end
fun @g {x} y = x + y
fun @h {} t = t 3

let f = clos @f {} in
let h = clos @h {} in
let t1 = f 2 in h t1 end end end
```

闭包转换（closure conversion）

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$
$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$
$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letfun}(f, x, E_1, E_2)$

$A ::= \mathsf{const}(n) \mid \mathsf{var}(x)$
$C ::= A \mid \mathsf{op}(A_1, A_2) \mid \mathsf{call}(A_1, A_2) \mid \mathsf{if}(A_1, E_2, E_3)$
$E ::= C \mid \mathsf{let}(x, C_1, E_2) \mid \mathsf{letclos}(f, @\ell, \{\overrightarrow{A_1, \cdots, A_m}\}, E_2)$
$P ::= E \mid \mathsf{fun}(@\ell, \{\overrightarrow{y_1, \cdots, y_m}\}, x, E), P$

- 针对每个 $\mathsf{letfun}(f, x, E_1, E_2)$，做如下处理：
  - 生成一个新的函数标记 $@\ell$
  - 分析函数体定义中的「自由变量」，以它们为闭包参数 $\overrightarrow{y_1, \cdots, y_m}$
  - 在最外层生成一个新的函数 $\mathsf{fun}(@\ell, \{\overrightarrow{y_1, \cdots, y_m}\}, x, E_1)$
  - 把原来的 letfun 位置转换为 $\mathsf{letclos}(f, @\ell, \{\overrightarrow{y_1, \cdots, y_m}\}, E_2)$
- **可以顺便做的优化：如果知道具体的调用函数并且该函数没有自由变量，就不用创建闭包**

闭包转换（closure conversion）

```
let makefac maker =
  let f x = if x == 0 then 1 else x * ((maker maker) (x - 1))
  in f end
in
  let fac = makefac makefac
  in fac 3 end
end
```

↓

```
let makefac maker =
  let f x =
    if x == 0 then 1 else
    let t1 = maker maker in
      let t2 = x - 1 in
        let t3 = t1 t2 in x * t3 end
      end
    end
  in f end
in
  let fac = makefac makefac
  in fac 3 end
end
```

闭包转换（closure conversion）

```
let makefac maker =
  let f x =
    if x == 0 then 1 else
    let t1 = maker maker in
      let t2 = x - 1 in
        let t3 = t1 t2 in x * t3 end
      end
    end
  in f end
in
  let fac = makefac makefac
  in fac 3 end
end
```

→

```
fun @makefac {} maker =
  let f = clos @f {maker} in f end

fun @f {maker} x =
  if x == 0 then 1 else
  let t1 = maker maker in
    let t2 = x - 1 in
      let t3 = t1 t2 in x * t3 end
    end
  end

let makefac = clos @makefac {} in
  let fac = makefac makefac in
    fac 3
  end
end
```

闭包转换（closure conversion）

```
fun @makefac {} maker =
  let f = clos @f {maker} in f end

fun @f {maker} x =
  if x == 0 then 1 else
  let t1 = maker maker in
    let t2 = x - 1 in
      let t3 = t1 t2 in x * t3 end
    end
  end

let makefac = clos @makefac {} in
  let fac = makefac makefac in
    fac 5
  end
end
```

```
function $makefac(env, maker) {
  var f = { func: $f, env: { maker: maker } };
  return f;
}

function $f(env, x) {
  if (x == 0)
    return 1;
  else {
    var t1 = env.maker.func(env.maker.env, env.maker);
    var t2 = x - 1;
    var t3 = t1.func(t1.env, t2);
    return x * t3;
  }
}

function main() {
  var makefac = { func: $makefac, env: {} };
  var fac = makefac.func(makefac.env, makefac);
  return fac.func(fac.env, 5);
}
```

每个函数都带一个环境参数，
记录其使用的自由变量

构造闭包时，记录函数指针
func 和其闭包环境 env

调用函数时，根据闭包的 func 取出函数指针，闭包的 env 取出其求值环境

主要内容

- 函数式编程语言 micro-ML
- 中间表示：若干种正规形式
- 前端到中端：源程序到正规形式的转换
- 中端到后端：正规形式程序的闭包转换
- **中端上的优化**

以 ANF 为中间表示

- 回顾：复写传播、常量传播、公共子表达式消除、死代码消除

复写传播

```
let x = y in x + y end
```

```
let x = y in y + y end
```

常量传播

```
let x = 3 in
  let y = 4 in x + y end
end
```

```
let x = 3 in
  let y = 4 in 3 + 4 end
end
```

常量折叠

```
let x = 3 in
  let y = 4 in 7 end
end
```

公共子表达式消除

```
let t = x + y in x + y end
```

```
let t = x + y in t end
```

死代码消除

```
let t = x + y in x - y end
```

```
x - y
```

以 ANF 为中间表示

- 回顾：复写传播、常量传播、公共子表达式消除、死代码消除

```
let y = 4 in
  let z = y in
    let t1 = x + y in
      let t2 = x + z in
        t1 * t2
      end
    end
  end
end
```

```
let y = 4 in
  let z = y in
    let t1 = x + y in
      let t2 = x + y in
        t1 * t2
      end
    end
  end
end
```

```
let y = 4 in
  let z = 4 in
    let t1 = x + 4 in
      let t2 = x + 4 in
        t1 * t2
      end
    end
  end
end
```

```
let y = 4 in
  let z = 4 in
    let t1 = x + 4 in
      let t2 = t1 in
        t1 * t2
      end
    end
  end
end
```

```
let y = 4 in
  let z = y in
    let t1 = x + 4 in
      let t2 = t1 in
        t1 * t2
      end
    end
  end
end
```

```
let t1 = x + 4 in
  t1 * t1
end
```

复写传播

- 对于 **let** x = y **in** exp **end**, 把 exp 中对 x 的使用替换为 y

产生规则	属性计算规则
$A \to \mathsf{const}(n)$	$A\,.\,\mathsf{anf} = \mathsf{const}(n)$
$A \to \mathsf{var}(x)$	$A\,.\,\mathsf{anf} = \mathsf{var}(A\,.\,\sigma(x))$
$C \to \mathsf{op}(A_1, A_2)$	$A_1\,.\,\sigma = A_2\,.\,\sigma = C\,.\,\sigma$ $\quad C\,.\,\mathsf{anf} = \mathsf{op}(A_1\,.\,\mathsf{anf}, A_2\,.\,\mathsf{anf})$
$C \to \mathsf{call}(A_1, A_2)$	$A_1\,.\,\sigma = A_2\,.\,\sigma = C\,.\,\sigma$ $\quad C\,.\,\mathsf{anf} = \mathsf{call}(A_1\,.\,\mathsf{anf}, A_2\,.\,\mathsf{anf})$
$C \to \mathsf{if}(A_1, E_2, E_3)$	$A_1\,.\,\sigma = E_2\,.\,\sigma = E_3\,.\,\sigma = C\,.\,\sigma$ $\quad C\,.\,\mathsf{anf} = \mathsf{if}(A_1\,.\,\mathsf{anf}, E_2\,.\,\mathsf{anf}, E_3\,.\,\mathsf{anf})$
$E \to \mathsf{let}(x, C_1, E_2)$	$C_1\,.\,\sigma = E\,.\,\sigma$ $\quad E_2\,.\,\sigma = [x \mapsto (C_1\,.\,\mathsf{anf}\ \mathsf{is}\ \mathsf{var}(y)\ ?\ y : x)]E\,.\,\sigma$ $E\,.\,\mathsf{anf} = \mathsf{let}(x, C_1\,.\,\mathsf{anf}, E_2\,.\,\mathsf{anf})$
$E \to \mathsf{letfun}(f, x, E_1, E_2)$	$E_1\,.\,\sigma = [x \mapsto x, f \mapsto f]E\,.\,\sigma$ $\quad E_2\,.\,\sigma = [f \mapsto f]E\,.\,\sigma$ $E\,.\,\mathsf{anf} = \mathsf{letfun}(f, x, E_1\,.\,\mathsf{anf}, E_2\,.\,\mathsf{anf})$

继承属性 σ 记录变量名到变量名的映射

如果形如 let x = y, 则在 σ 中记录该复写

常量传播

- 对于 **let** x = *INT* **in** exp **end**, 把 exp 中对 x 的使用替换为 *INT*
- 对于 *INT* op *INT*, 替换为运算结果(即常量折叠)

继承属性 σ 记录变量名到常量的映射

如果运算分量都是常量则折叠

不是常量的变量要移除

产生规则	属性计算规则
$A \to \mathsf{const}(n)$	$A.\mathsf{anf} = \mathsf{const}(n)$
$A \to \mathsf{var}(x)$	$A.\mathsf{anf} = x \in A.\sigma \ ?\ \mathsf{const}(A.\sigma(x)) : \mathsf{var}(x)$
$C \to \mathsf{op}(A_1, A_2)$	$A_1.\sigma = A_2.\sigma = C.\sigma$ $C.\mathsf{anf} = A_1.\mathsf{anf}\ \mathsf{is}\ \mathsf{const}(n_1) \wedge A_2.\mathsf{anf}\ \mathsf{is}\ \mathsf{const}(n_2)\ ?$ $\mathsf{const}(f_{\mathsf{op}}(n_1, n_2)) : \mathsf{op}(A_1.\mathsf{anf}, A_2.\mathsf{anf})$
$C \to \mathsf{call}(A_1, A_2)$	$A_1.\sigma = A_2.\sigma = C.\sigma$ $C.\mathsf{anf} = \mathsf{call}(A_1.\mathsf{anf}, A_2.\mathsf{anf})$
$C \to \mathsf{if}(A_1, E_2, E_3)$	$A_1.\sigma = E_2.\sigma = E_3.\sigma = C.\sigma$ $C.\mathsf{anf} = \mathsf{if}(A_1.\mathsf{anf}, E_2.\mathsf{anf}, E_3.\mathsf{anf})$
$E \to \mathsf{let}(x, C_1, E_2)$	$C_1.\sigma = E.\sigma$ $E_2.\sigma = C_1.\mathsf{anf}\ \mathsf{is}\ \mathsf{const}(n)\ ?\ [x \mapsto n]E.\sigma : E.\sigma \backslash \{x\}$ $E.\mathsf{anf} = \mathsf{let}(x, C_1.\mathsf{anf}, E_2.\mathsf{anf})$
$E \to \mathsf{letfun}(f, x, E_1, E_2)$	$E_1.\sigma = E.\sigma \backslash \{f, x\}$ $E_2.\sigma = E.\sigma \backslash \{f\}$ $E.\mathsf{anf} = \mathsf{letfun}(f, x, E_1.\mathsf{anf}, E_2.\mathsf{anf})$

公共子表达式消除

- 对于 **let** x = a op b **in** exp **end**，把 exp 中相同的 a op b 替换为 x，需要注意 a 和 b 是变量时要考虑变量覆盖问题

继承属性 σ 记录表达式到变量的映射

产生规则	属性计算规则
$C \to \mathsf{op}(A_1, A_2)$	$C.\mathsf{anf} = \mathsf{op}(A_1, A_2) \in C.\sigma\ ?\ C.\sigma(\mathsf{op}(A_1, A_2)) : \mathsf{op}(A_1, A_2)$
$C \to \mathsf{call}(A_1, A_2)$	$C.\mathsf{anf} = \mathsf{call}(A_1, A_2)$
$C \to \mathsf{if}(A_1, E_2, E_3)$	$E_2.\sigma = E_3.\sigma = C.\sigma$ $\quad C.\mathsf{anf} = \mathsf{if}(A_1, E_2.\mathsf{anf}, E_3.\mathsf{anf})$
$E \to \mathsf{let}(x, C_1, E_2)$	$C_1.\sigma = E.\sigma$ $E_2.\sigma = C_1.\mathsf{anf}\ \mathsf{is}\ \mathsf{op}(a, b)\ ?\ [\mathsf{op}(a, b) \mapsto x](E.\sigma - x) : (E.\sigma - x)$ $E.\mathsf{anf} = \mathsf{let}(x, C_1.\mathsf{anf}, E_2.\mathsf{anf})$
$E \to \mathsf{letfun}(f, x, E_1, E_2)$	$E_1.\sigma = E.\sigma - f - x$ $\quad E_2.\sigma = E.\sigma - f$ $E.\mathsf{anf} = \mathsf{letfun}(f, x, E_1.\mathsf{anf}, E_2.\mathsf{anf})$

去掉之前记录的涉及 x 的表达式

为了简化变量名的处理，可以在优化前改写一遍所有变量名，使得它们都不相同

死代码消除

- 对于 **let** x = ... **in** exp **end**，若 exp 中没有使用 x，则可以把这个 **let** 绑定消除

综合属性 fv 记录出现的自由变量的集合

产生规则	属性计算规则	
$A \to \mathsf{const}(n)$	$A\,.\,\mathsf{fv} = \varnothing$	
$A \to \mathsf{var}(x)$	$A\,.\,\mathsf{fv} = \{x\}$	
$C \to \mathsf{op}(A_1, A_2)$	$C\,.\,\mathsf{fv} = A_1\,.\,\mathsf{fv} \cup A_2\,.\,\mathsf{fv}$	$C\,.\,\mathsf{anf} = \mathsf{op}(A_1, A_2)$
$C \to \mathsf{call}(A_1, A_2)$	$C\,.\,\mathsf{fv} = A_1\,.\,\mathsf{fv} \cup A_2\,.\,\mathsf{fv}$	$C\,.\,\mathsf{anf} = \mathsf{call}(A_1, A_2)$
$C \to \mathsf{if}(A_1, E_2, E_3)$	$C\,.\,\mathsf{fv} = A_1\,.\,\mathsf{fv} \cup E_2\,.\,\mathsf{fv} \cup E_3\,.\,\mathsf{fv}$	$C\,.\,\mathsf{anf} = \mathsf{if}(A_1, E_2\,.\,\mathsf{anf}, E_3\,.\,\mathsf{anf})$
$E \to \mathsf{let}(x, C_1, E_2)$	$E\,.\,\mathsf{anf} = x \in E_2\,.\,\mathsf{fv}\ ?\ \mathsf{let}(x, C_1\,.\,\mathsf{anf}, E_2\,.\,\mathsf{anf}) : E_2\,.\,\mathsf{anf}$ $E\,.\,\mathsf{fv} = x \in E_2\,.\,\mathsf{fv}\ ?\ (C_1\,.\,\mathsf{fv} \cup (E_2\,.\,\mathsf{fv} \backslash \{x\})) : E_2\,.\,\mathsf{fv}$	
$E \to \mathsf{letfun}(f, x, E_1, E_2)$	$E\,.\,\mathsf{anf} = f \in E_2\,.\,\mathsf{fv}\ ?\ \mathsf{letfun}(f, x, E_1\,.\,\mathsf{anf}, E_2\,.\,\mathsf{anf}) : E_2\,.\,\mathsf{anf}$ $E\,.\,\mathsf{fv} = f \in E_2\,.\,\mathsf{fv}\ ?\ ((E_1\,.\,\mathsf{fv} \backslash \{f, x\}) \cup (E_2\,.\,\mathsf{fv} \backslash \{f\})) : E_2\,.\,\mathsf{fv}$	

如果 x 不出现在 E_2 使用的自由变量里则消除

E_1 是函数体定义，其内部使用的函数名 f 和参数名 x 不是自由变量

参考文献

- Eijiro Sumii. 2005. MinCaml: a simple and efficient compiler for a minimal functional language. In Proceedings of the 2005 workshop on Functional and declarative programming in education (FDPE '05).
- Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The essence of compiling with continuations. In Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93).
- Matt Might. By example: Continuation-passing style in JavaScript.