中间代码

为什么要使用中间代码

- **补齐源代码和机器码之间的抽象级别差异，方便代码优化**
- 源代码：抽象层级高，有类型和复杂的语法语义定义
- 机器码：只有寄存器和简单的指令
- 需要某种东西把两者接起来

中间代码要做成什么样子

没有固定的套路，但是：

- **将复杂的语义拆成简单的部分，以便进行分析、修改、优化**
- 比源代码抽象层级低
  - 消除源代码中的语法糖
  - 补充一些源代码中没有写出的信息（比如具体的类型信息）
- 比机器码抽象层级高
  - 可以使用更高层的类型系统和数据结构
  - 通常拥有大量（无限）的寄存器

宗旨：好用就行

常见的中间代码形式

命令式 (imperative) 语言

通常基于语句和控制流构建，变量通常可变。

- 三地址码/四元式
- SSA (Static Single Assignment/静态单赋值)

函数式 (functional) 语言

通常基于表达式和函数调用构建，变量通常不可变。

- A-Normal Form / K-Normal Form
- Continuation-passing Style

命令式的中间代码

三地址码

给表达式中的每个计算步骤都起一个名字，把不跳转的代码区间组织成基本块。

```
x = a + b
y = x + 3
x = x + y
```

SSA

变量不能被重新赋值，更便于进行数据流分析。

```
x1 = a + b
y1 = x1 + 3
x2 = x1 + y1
```

函数式的中间代码

A-Normal Form (ANF)

强制要求表达式的每个计算步骤都起一个名字，表达式中只能出现名字不能出现其他表达式。

```
let x = a + b in
let y = x + 1 in
let cond = y > 0 in
if cond  then (x)  else (0)
```

- 便于简化控制流和数据流分析。

K-Normal Form (KNF) 与 ANF 基本一致，只是规范化程度有差别，后面不做区分。

函数式的中间代码

Continuation-Passing Style (CPS)

反转程序的控制流，把所有之后要执行的东西打包成一个部分 ("Continuation") 作为参数传递。

```
add'(a, b, K1) where K1 = x =>
  add'(x, 1, K2) where K2 = y =>
    gt'(y, 0, K3) where K3 = cond =>
      if cond then RET(x) else RET(0) where
        RET -- returns to the enclosing environment
```

- 方便跨函数和特殊控制流（如异常）的优化，某些时候会很有用。
- 在代码中的表示和分析相对来说难一些。
- （现代一些的 CPS 会只用 continuation 表示控制流）

K-Normal Form

Expression:

$$
\begin{aligned}
e ::= {} & n && \text{(name)} \\
& \mid \text{int}(i) \mid n + n \mid n - n \mid \cdots && \text{(simple expressions)} \\
& \mid \text{let } n : T = e \text{ in } e_{cont} && \text{(bind names)} \\
& \mid \text{let rec } f(x, y, z) = e_{body} \text{ in } e_{cont} && \text{(create functions)} \\
& \mid \text{if } n \text{ then } e_{true} \text{ else } e_{false} && \text{(conditionals)}
\end{aligned}
$$

从 AST 构造 KNF

递归访问每个子表达式，给每个子表达式起一个名字

```
a * b + c * d
```

```
Add( Mul( Var(a), Var(b) ),
     Mul( Var(c), Var(d) ) )
```

```
let _1 = a * b;
let _2 = c * d;
_1 + _2
```

从 AST 构造 KNF

```
fn expr_to_knf(env, e: Expr) -> Knf {
  match e {
    Var(x) => Knf::Var(x)
    Add(a, b, ty) => {
      let a_name = generate_name()                    // _1
      let b_name = generate_name()                    // _2
      Knf::Let(a_name, ty, expr_to_knf(env, a),       // let _1 = <a>
        Knf::Let(b_name, ty, expr_to_knf(env, b),     // let _2 = <b>
          Knf::Add(a_name, b_name)))                  // _1 + _2
    }
    Let(x, t, expr, cont) => {
      let new_env = env.add(x, t)
      Knf::Let(x, t, expr_to_knf(env, expr),          // let x: T = <expr> in
        expr_to_knf(new_env, cont))                   // <cont>
    }
  }
}
```

KNF 上可以做什么优化?

典型的编译优化基本都可以做:

- 常量折叠
- 函数内联
- 死代码删除
- 公共子表达式合并
- 运算强度削弱
- 循环（闭包）不变量外提
- etc.

怎么做优化?

用 `match` 匹配我们感兴趣的变体，然后直接构造新的 KNF 结构。

```
fn optimize(e: Knf) -> Knf {
  match e {
    // Interesting
    Mul(x, y) => {
      if interesting_constraint(x, y) {
        let (a, b) = computation(x, y)
        Add(a, b)  // return new KNF structure
      } else {
        e  // Not intersting, don't modify
      }
    }
    // Not intersting, don't modify
    _ => e
  }
}
```

前置操作：α-conversion (Alpha 变换)

变量名并不重要，但是在剪接代码的时候不要改变变量名指向的定义

$\rightarrow$ **给每一个变量都取一个独一无二的名字**

```
fn alpha(existing, e: Knf) -> Knf {
  match e {
    Let(mut x, t, e1, mut cont) => {
      if existing.contains(x) {     // 有重名
        let new_x = generate()      // 起个新名字
        cont = replace(x, new_x, cont) // 替换后面所有用到的地方
        x = new_x
      }
      existing.add(x)               // 记下变量有定义
      Let(x, t, alpha(cont))          // 递归进行操作
    }
    _ => e                          // 剩下的都不用管
  }
}
```

常量折叠

将参数都是常量的运算在编译期直接计算成结果。

- `let x = 3 in (x + 5)` $\longrightarrow$ `(3 + 5)` $\longrightarrow$ `8`
- **通常用在数学运算上**，但其实只要能在编译期间算出来的东西都能用
- 一个更强的变体叫做部分求值（Partial Evaluation）

常量折叠

```
fn const_folding(constants, e: Knf) -> Knf {
  match e {
    Let(x, t, e1, cont) => {
      let e1 = const_folding(constants, e1)          // 先对被赋值的表达式进行折叠
      if is_constant(e1) { constants.add(x, e1) }    // 记录常量
      let cont = const_folding(constants, cont)      // 再用新信息对后续的代码进行折叠
      Let(x, t, e1, cont)
    }
    Add(x, y) => {
      match (constants.get(x), constant.get(y)) {
        (Some(i), Some(j)) => Int(i + j)             // 两边都是常量，直接计算结果
        _ => e                                       // 不是则不变
      }
    }
    _ => e
  }
}
```

函数内联

把比较小的函数的调用替换成函数体本身，节省调用开销。

`let rec f(x) = x + 1 in f(z)` => `z + 1`

- 由于已经做过 Alpha 变换，所以不用担心函数体中的变量和参数重名
- 结束后函数可能被内联到自己中，所以需要再做一次 Alpha 变换

函数内联

```
fn inline(functions, e: Knf) -> Knf {
  match e {
    LetRec(f, t, body, cont) => { functions.add(f); e }
    Call(f, args) => {
      if functions.get(f).size() < threshold {  // 函数比较小
        f.body.replace(f.formal_args, args)     // 替换函数调用为函数体，替换形参为实参
      } else {
        e
      }
    }
  }
}
```

死代码删除

如果一个变量没有在后面用到，也没有副作用，那么可以直接删掉。

- 副作用：对外部环境的修改，比如调用 print 函数，或者写入数组

```
fn dce(e: Knf) -> Knf {
  match e {
    Let(x, t, e1, cont) => {
      let e1 = dce(e1)
      if !cont.uses(e1) &&          // 后面没用
         !e1.has_side_effects() { // 没副作用
        dce(cont)                  // 替换成后面的代码
      } else {
        Let(x, t, e1, dce(cont))
      }
    }
    _ => e
  }
}
```

公共子表达式合并

如果一个表达式之前已经算过了，也没有副作用，那么可以把它替换成之前的结果。

```
fn cse(exprs, e: Knf) -> Knf {
  match e {
    Let(x, t, e1, cont) => {
      let e1 = cse(exprs, e1)
      match exprs.get(e1) {
        Some(name1) => replace(x, name1, cont)
        _ => Let(x, t, e1, cont)
      }
    }
    _ => e
  }
}
```

- 为了优化性能，反过来也是可能的：Rematerialization（再算一遍比加载更快）

运算强度削弱

把代价较高的运算替换成等效的、代价更低的运算。

- 把 2 的幂的乘除法转换成左右移：`x * 2` => `x << 1`
- 把与常量的乘法转换成左右移和加减法的结合：`x * 15` => `x << 4 - x`
- 把与常量的除法、模除转换成乘法和加减法的结合：
  `(uint) x / 3` => `(uint64_t) x * 0xAAAA_AAAB >> 33`
  Torbjorn Granlund, et al. Division by Invariant Integers using Multiplication. 1994

循环/闭包不变量外提

对于一个被调用多次的代码块（函数/循环），如果一个运算没有副作用，且不依赖函数输入/循环变量，可以把它提到函数/循环以外。

- MiniMoonBit 并没有循环，但是对于递归函数可以做类似的事情

```
let a = 1
for ... {
  let x = a + 1  // ← 这个表达式不涉及循环变量
  deal_with(x)
}
```

```
let a = 1
let x = a + 1    // ← 提出到循环以外
for ... {
  deal_with(x)
}
```