

1 机器指令

在第一章中，我们已经简单地讨论了 C 语言如何在内存上存放数据，包括整数类型 (`uint64_t`)，浮点数类型 (`float`)，指针类型 (`void *`)。实际上，在程序中不同位置的数据会被存放在虚拟内存中的不同位置，这一点我们会在讨论链接时进一步讨论。现在，我们利用基本数据类型，特别是整数类型，就可以构造一个可计算的模型了。

对于计算机的计算能力，我们从一个理想的数学上的计算模型出发。这个理想的计算机被称为无界贮存机 (Unlimited Register Machine, URM)，URM 拥有无限数量的贮存器¹： R_1, R_2, R_3, \dots ，如图1。在任意时刻，每一个贮存器 R_i 上储存一个自然数 $r_i \in \mathbb{N}_0$ 。对于 URM，我们定义四种指令：



图 1: 无界贮存机 URM

1. 置零指令 (Zero): $Z(i) : r_i \leftarrow 0$, 该指令将贮存器 R_i 的值 r_i 置为零;
2. 后继指令 (Successor): $S(i) : r_i \leftarrow r_i + 1$, 该指令将贮存器 R_i 的值 r_i 增加 1;
3. 移动指令 (Transfer): $T(h, i) : r_i \leftarrow r_h$, 该指令将贮存器 R_i 的值 r_i 替换为贮存器 R_h 的值 r_h ;
4. 跳转指令 (Jump): $J(h, i, j)$, 该指令并不是对贮存器的操作，而是对指令序列的操作。在指令序列 $I_{i-1, 2, \dots} \in \{Z, S, T, J\}$ 中，URM 按照顺序执行指令。当执行到跳转指令 $I_h = J(h, i, j)$ 时，URM 根据贮存器 R_h 的值 r_h 与贮存器 R_i 的值 r_i 决定下一步指令：如果有 $r_h = r_i$ ，则跳转到 j 指令 I_j ；如果有 $r_h \neq r_i$ ，则继续执行下一条指令 I_{h+1} ；

这样，我们就可以开始讨论计算机的计算能力了。上述对计算模型的描述其实隐含了计算机的物理实现，但也有不少差别。URM 的贮存器可以被视为物理内存上的每一个内存位置，但每一个内存位置并不足以储存整个自然数集 \mathbb{N}_0 。如果我们按照 `uint64_t` 的数据类型去对齐内存，那么每一个贮存器最多可以计数到 $2^{64} - 1$ ，也足够大了。除此以外，URM 只有四种指令，指令的数量远远小于我们所常见的指令集 (Instruction Set Architecture)。但是我们会发现，URM 拥有无限的贮存容量，这使得它的表达能力足够丰富²。还有一点就是 URM 的指令是独立的，并没有被储存在贮存器上。

URM 的四种指令使得我们立即可以实现三种基本函数 (Function)。如果我们将第一个贮存器 R_1 默认为保存程序 (指令序列) 运行结果的贮存器，那么三种基本函数可以通过 URM 指令直接实现：

1. 置零函数 (Zero): $z : \mathbb{N}_0 \ni x \mapsto 0$, URM 实现为 $Z(1)$;
2. 后继函数 (Successor): $s : \mathbb{N}_0 \ni x \mapsto x + 1 \in \mathbb{N}_0$, URM 实现为 $S(i)$;
3. 投影函数 (Projection): $u : \mathbb{N}_0^n \ni x = [x_1, x_2, \dots, x_n] \mapsto x_i \in \mathbb{N}_0$, URM 实现为 $T(i, 1)$;

除此以外，显然跳转指令无法直接建立函数，因为它并不读写贮存器，而是控制执行指令的流程。

我们之所以要建立 URM 指令和函数之间的关系，是为了通过函数去研究指令序列，也就是程序。因为我们无法直接研究程序，但却可以描述函数的性质。在建立了 URM 指令和函数之间的关系以后，我们

¹Register 应当被翻译为寄存器，但物理计算机的 CPU 中包含了寄存器 (Register)。为了区分，我们称理想计算机中的 Register 为“贮存器”。

²关于这一点，同时我们注意到，在有限的指令范围内，通常所涉及的贮存容量也是有限的，甚至是相邻近的。因此我们讨论一个可停机 (Halt) 的程序时，总是在有限的贮存容量上讨论，这是局部性的一种体现。

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>
计算机系统

可以进一步建立 URM 程序（指令序列）和其他函数之间的关系。显然，URM 的任一程序 P ，是按照顺序（或跳转）执行 URM 的四条指令实现的，我们假定 P 接受 n 元输入 $\mathbf{x} = [x_1, \dots, x_n] \in \mathbb{N}_0^n$ ，将计算结果保存在默认的贮存器 R_1 上，那么程序 P 其实是一个数学函数的具体实现： $f_P : \mathbb{N}_0^n \ni \mathbf{x} \mapsto r_1 \in \mathbb{N}_0$ 。

可以证明，任一程序 P 所实现的函数 $f_P : \mathbb{N}_0^n \ni \mathbf{x} \mapsto r_1 \in \mathbb{N}_0$ ，都可以通过对三种基本函数 z, s, u 进行有限次操作生成（Generate）。在这里，一共有三种操作，或者说对函数的映射，算子（Operator）：

替换，或复合（Substitution）

对于 \mathbb{N}_0^n 到 \mathbb{N}_0 的一系列函数 g_1, g_2, \dots, g_m ：

$$g_1 : \mathbb{N}_0^n \ni \mathbf{x} \mapsto y_1 \in \mathbb{N}_0, \dots, g_m : \mathbb{N}_0^n \ni \mathbf{x} \mapsto y_m \in \mathbb{N}_0$$

以及外层函数 $h : \mathbb{N}_0^m \ni [y_1, \dots, y_n] \mapsto t \in \mathbb{N}_0$ ，则算子 σ 生成复合函数：

$$\sigma : (h, g_1, g_2, \dots, g_m) \mapsto (\mathbf{x} \mapsto h(g_1(\mathbf{x}), \dots, g_m(\mathbf{x})) \in \mathbb{N}_0)$$

其中函数 h, g_1, \dots, g_m 都是 z, s, u 所生成的。

递归，或基本递归（Recursion）

对于 \mathbb{N}_0^n 到 \mathbb{N}_0 的初始函数 $f : \mathbb{N}_0^n \ni \mathbf{x} \mapsto f(\mathbf{x}) \in \mathbb{N}_0$ ，以及递归过程 $g : \mathbb{N}_0^n \ni \mathbf{x}, \mathbb{N}_0 \ni t, \mathbb{N}_0 \ni y \mapsto g(\mathbf{x}, t, y) \in \mathbb{N}_0$ ，算子 ρ 生成基本递归函数 $h : \mathbb{N}_0^n \ni \mathbf{x}, \mathbb{N}_0 \ni t \mapsto h(\mathbf{x}, t) \in \mathbb{N}_0$ ：

$$\begin{aligned} h(\mathbf{x}, 0) &= f(\mathbf{x}) \\ h(\mathbf{x}, t+1) &= g(\mathbf{x}, t, h(\mathbf{x}, t)) \end{aligned}$$

其中 t 为递归的计数器，计数当前的递归层次。这样，算子 ρ 生成了一系列不同递归层次的函数：

$$\rho : (f, g) \mapsto (h : \mathbb{N}_0^n \ni \mathbf{x}, \mathbb{N}_0 \ni t \mapsto h(\mathbf{x}, t) \in \mathbb{N}_0)$$

其中函数 f, g 都是 z, s, u 所生成的。

最小化，或搜索（Minimalisation）

对函数 $f(\mathbf{x}, y)$ ，固定参数 \mathbf{x} 后，即形成了有关变量 y 的函数 $f : \mathbb{N}_0^n \ni \mathbf{x}, \mathbb{N}_0 \ni y \mapsto f(\mathbf{x}, y) \in \mathbb{N}_0$ 。对于变量 $y \in \mathbb{N}_0$ ，我们从 0 开始搜索： $y = 0, 1, 2, \dots$ ，直到第一个满足下列条件的 y ：

1. $f(\mathbf{x}, y) = 0$ ，也即 y 是第一个（最小的）满足约束条件的值；
2. $(\forall t = 0, 1, \dots, y-1) ((f(\mathbf{x}, t) \uparrow) \wedge (f(\mathbf{x}, t) \neq 0))$ ，其中符号 \uparrow 表示函数在该点上未定义（Undefined）， \downarrow 表示有定义（Defined）。这要求 y 以前的所有值 t 在函数 $f(\mathbf{x}, t)$ 有定义，并且不满足约束条件；

这样，我们搜索到满足约束条件的最小 y 值，即为 μ 算子所生成的函数，记作 $\mu_y^{f(\mathbf{x}, y)=0}$ ：

$$\mu : f \mapsto \mu_y^{f(\mathbf{x}, y)=0} = \begin{cases} y_{\min} & \exists y_{\min} \\ \uparrow & \nexists y_{\min} \end{cases}$$

其中，函数 f 由 z, s, u 所生成。

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min> 计算机系统

另一方面，也可以证明，由 z, s, u 三个基本函数，经过任意有限次 σ, ρ, μ 操作，所生成的函数 f ，都存在一个 URM 程序 P_f 可以实现它的计算。这样，任意有限长度的 URM 程序 ($P = I_1^P, I_2^P, \dots$) 所形成的集合 \mathcal{P} ，也对应了基本函数 z, s, u 在 σ, ρ, μ 操作下生成的集合 \mathcal{R} ，这两个集合是等价的： $\mathcal{P} \subseteq \mathcal{R}$, $\mathcal{R} \subseteq \mathcal{P}$ 。不仅如此，还可以证明生成函数的集合是闭集合 (Closed Set)：

$$\begin{aligned}\mathcal{R} &\ni z, s, u \\ \mathcal{R} &\ni h, g_1, \dots, g_m, \quad \sigma(h, g_1, \dots, g_m) \in \mathcal{R} \\ \mathcal{R} &\ni f, g, \quad \rho(f, g) \in \mathcal{R} \\ \mathcal{R} &\ni f, \quad \mu(f) \in \mathcal{R}\end{aligned}$$

函数类 \mathcal{R} 被称为部分递归函数 (Partial Recursive Function)。现在，我们（其实是 Alonzo Church）对部分递归函数与可计算性提出一个论断，这被称为 Church 论题 (Church Thesis)，或被称为 Church-Turing 论题。在 URM 的语境下，Church 论题认为：部分递归函数类 \mathcal{R} 是“可计算”的函数，URM 所得的程序是“可计算”的程序。

关于这一点，我们进行类比，方便读者理解。例如在分析学中，对于“极限 (Limit)”存在感性的理解与严格的描述，也就是 Weierstrass 提出的 $\epsilon - \delta$ 语言。那么，我们如何能确定 $\epsilon - \delta$ 语言所描述的性质，正是“极限”呢？实际上， $\epsilon - \delta$ 语言所描述的要比“极限”的含义更加丰富。对于 Church 论题与“可计算”也同样如此。函数类 \mathcal{R} 是否是对“可计算”的一个恰当描述，关于这一点我们无从确定。但是由于目前经验证据相当充足，因此我们认为用 \mathcal{R} 去描述“可计算”是合理的。

在 URM 中，我们看到了两个重要的元素：被储存在贮存器中的数据，以及操纵数据的指令。因此，正如我们先前所说的，URM 的描述中隐含了计算机的物理模型。下面，我们就要开始逐步在现实世界中实现这个理想的可计算模型了。实际上，在我们对计算机的实现中，所有的指令都与 URM 的四条指令有千丝万缕的联系。而部分递归函数的三个算子，都可以在 URM 中得到实现： σ 算子的复合函数通过栈上的过程调用实现； ρ 算子的基础递归可以通过循环实现，即条件判断与指令跳转，或者通过过程调用； μ 算子的搜索可以通过循环实现。

1.1 指令的硬件资源

现在，我们开始思考对 URM 的实现，首先要考虑的就是指令所需要的硬件 (Hardware) 资源。首先，我们需要解决的是指令本身的储存问题。在 URM 中，指令是无处储存的，URM 只是不停地执行计算，却不知晓指令从何处取得。在 Von Neumann 的体系结构中，指令本身被视作特殊的数据，也被储存在内存上。关于这一点，我们认为是 Von Neumann 体系结构的一个重大特点。

“指令即数据”，这是非常深遂的思想。基于这一思想，在目前常见的计算机系统中，程序和代码是被保存在持久性储存设备上的数据，也仍是以 Byte 为单位的数据。程序在计算机系统中运行时，指令会被加载到物理内存中，被标注为只读数据，这样就实现了对指令代码的保护。CPU 按照指向指令的指针读取当前指令，然后根据指令将数据从物理内存加载到 CPU，进行计算，并且将计算结果写回到 CPU 和内存。这就是一条指令所能调度的主要资源。

基于计算程序与部分递归函数的等价性，我们也可以将 Von Neumann 的思想理解为“函数即数据”。反过来，“数据即函数”，这一思想也同样深刻，充满洞见。为了描述这一点，我们采用 Church 研究的 λ 演算 (λ -Calculus)。特别注意， λ 演算是与 URM 在递归函数上等价的计算模型。 λ 表达式的语法规则如下，假定 E 是一个 λ 表达式， Var 是无法被继续展开的变量，那么 E 有三种展开：

$$E \rightarrow \text{Var} \mid (\lambda \text{Var}.E) \mid (EE)$$

根据 λ 表达式, Church 将所有的自然数看作加法的归纳, 这样被定义的自然数被称为**Church 数**(Church Numerals), 它背后的原理是**Peano 公理**(Peano Axiom)。Church 没有定义数据 0, 而直接定义零函数和加法。考虑 $n + 0 = n$, 因此 0 对符号 + 是自我相等的。这样, 我们就可以定义零函数, 它不增加函数的阶数, 用 λ 表达式描述为 $(\lambda f.(\lambda x.x))$ 。

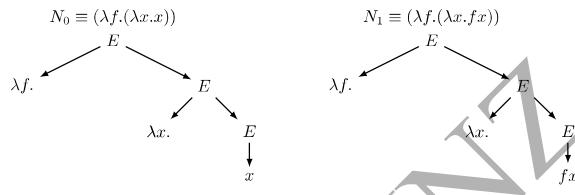


图 2: 0 (左) 与 1 (右)

λ 表达式对可替换的变量进行抽象。例如, 我们在 $(\lambda x.x^2)$ 中将符号 x 替换为 2, 这样我们就要将 λ 表达式归约到符号 $2^2 = 4: (\lambda x.x^2)[2] =_{\beta} 2^2 = 4$ 。这种替换被称为 β 归约(β Reduction), 记作 $=_{\beta}$ 。接下来, 我们定义 1 为 $N_1 \equiv (\lambda f.(\lambda x.fx))$ 。1 不再被视为一个数字, 而被视为归纳的过程, 因此它是对函数的运算, 增加函数的阶数。这样, 我们就可以归纳所有的自然数了。此时, 函数不再被视为数据, 相反, 数据被视为函数的抽象。

中央处理器(Central Processing Unit, CPU)中包含了寄存器组, 程序计数器, 以及条件码。我们依次观察它们的结构。以 64 位的 CPU 为例, 它的寄存器组一般包括通用的寄存器(Register), 用于: 保存返回值(Return Value), 保存过程调用的参数(Argumen), 描述过程调用的帧(Frame)、栈(Stack)结构, 如图3。

63	31	15	7	0
%rax	%eax	%ax	%al	
%rbx	%ebx	%bx	%bl	
%rcx	%ecx	%cx	%cl	
%rdx	%edx	%dx	%dl	
%rsi	%esi	%si	%sil	
%rdi	%edi	%di	%dil	
%rbp	%ebp	%bp	%bp1	
%rsp	%esp	%sp	%spl	

图 3: 前八个通用寄存器

注意到, `%rax` 寄存器在这里的作用, 与我们在 URM 模型中所默认的第一个贮存器 R_1 是相似的。保存参数的寄存器包括 `%rsi`, `%rdi` 等, 它们也与 URM 中的贮存器有相似的作用。这些内容我们到具体指令里去讨论, 目前我们只需要知道: URM 模型中所使用的贮存器, 一部分用来存储数据(内存实现), 另一部分用来操控指令, 而在后者中, 有部分贮存器被实现在通用寄存器组中。

除此了通用的寄存器外, CPU 还维护一个程序计数器 (Program Counter), 指向将要从物理内存(实际上是 L1 指令缓存)中被读取的下一条指令的虚拟地址。通常, 我们用 `%rip` 或 `%eip` 寄存器存放指令的虚拟地址。这个寄存器正是运用了将函数视为数据的思想。

在我们对计算机系统的近似当中, 我们可以用 C 语言的 `struct` 与 `union` 来模拟寄存器的结构。可以看到, `rax`, `eax` 等寄存器的低位部分是相同的, 因此我们可以用 `union` 使它们在模拟中共用低地址:

```
1  typedef struct
2  {
3      union
4      {
5          uint64_t rax;
6          uint32_t eax;
7          uint16_t ax;
8          struct
9          {
10             uint8_t al;
11             uint8_t ah;
12         };
13     };
14     // ...
15 } cpu_reg_t;
16 cpu_reg_t cpu_reg;
```

/src/headers/cpu.h


```
1  typedef union
2  {
3      uint64_t rip;
4      uint32_t eip;
5 } cpu_pc_t;
6 cpu_pc_t cpu_pc;
```

/src/headers/cpu.h

同时, 为了实现条件分支和 `rip` 跳转, CPU 维护一组标志寄存器 (Flags) 或条件码 (Conditional Codes), 用来记录上一条指令计算结果的信息。上一条指令通常是整数的算术运算: `inc`, `dec`, `neg`, `not`, `add`, `sub`, `imul`, `xor`, `and`, `sal`, `shl`, `sar`, `shr`, 或是比较: `cmp`, `test`。假定源操作数为 `src`, 目的操作数为 `dst`, 运算的结果为 `val`:

- CF: 将指令视为对无符号整数的运算, 判断运算是否发生无符号整数的溢出, 也即最高位发生进位 (Carry)。对于无符号加法, 即 $CF = (dst + src < dst)$, 对于无符号减法, 即 $CF = (dst - src > dst)$

- ZF: 判断指令运算的结果是否为 0, 即 $ZF = (val == 0)$
- SF: 判断运算结果的最高位是否为 1, 也即将结果视作有符号数, 判断其是否为负数, 即 $SF = (val >> 63)$
- OF: 视作有符号数的运算, 判断是否发生正数或负数的溢出。取三个数的符号位 `sign_src`, `sign_dst`, `sign_val`, 对于有符号加法, 即 $OF = !(sign_dst \wedge sign_src) \&& (sign_src \wedge sign_val)$, 对于有符号减法, 即 $OF = ((sign_dst \wedge sign_src) \&& !(sign_src \wedge sign_val))$

对于标志位, 考虑到 C 语言的数据对齐 (Alignment), 我们不必按照 Bit 级别进行模拟。同样使用 `union` 与 `struct`, 可以使我们在模拟中达到按 Bit 操作的效果:

```
/src/headers/cpu.h

1 // the 4 flags be a uint64_t in total
2 typedef struct
3 {
4     union
5     {
6         uint64_t __flags_value;
7         struct
8         {
9             uint16_t CF;
10            uint16_t ZF;
11            uint16_t SF;
12            uint16_t OF;
13        };
14    };
15 } cpu_flags_t;
16 cpu_flags_t cpu_flags;
```

在 CPU 之外, 我们还有用动态随机访问内存 (Dynamic Random Access Memory, DRAM) 实现的物理内存 (Physical Memory)。对于物理内存, 正如我们反复提及的, 其本质上可以被看作一个巨大的 Byte 数组。我们近似实现一个 64KB 大小的物理内存:

```
/src/headers/memory.h

1 #define PHYSICAL_MEMORY_SPACE 65536
2 uint8_t physical_mem[PHYSICAL_MEMORY_SPACE];
```

我们注意到, 物理内存的大小是有限的, 而虚拟地址的寻址范围远要比物理内存更宽泛。为了解决这个矛盾, 我们会在后文中进行地址翻译。为了方便实现这一点, 我们严禁直接对物理内存 `physical_mem` 进行读写, 而一定要使用下列用于内存访问的接口函数。这是我们对内存进行的抽象:

```
/src/headers/memory.h

1 // r/w uint64_t to DRAM
2 uint64_t    read64bits_dram   (uint64_t paddr);
3 void       write64bits_dram  (uint64_t paddr, uint64_t data);
```

```

4
5 // r/w char (*) [64] to DRAM
6 void      readinst_dram     (uint64_t paddr, char *buf);
7 void      writeinst_dram   (uint64_t paddr, const char *str);

/src/headers/cpu.h

1 uint64_t va2pa(uint64_t vaddr);

```

利用这些硬件资源，我们就可以对 CPU 的指令周期（Instruction Cycle）进行近似了，如图4。这也是理论计算机的一种实现，更具体的，是 Von Neumann 体系结构的一种实现。图4中所描述的指令周期与现实中的计算机结构有许多差别，我们的近似忽略了许多硬件同步的细节，而专注在指令执行的过程。这也符合我们对计算机系统近似的抽象层次。尽管如此，图4中依然包括了许多细节，我们在后续内容中逐个讨论。

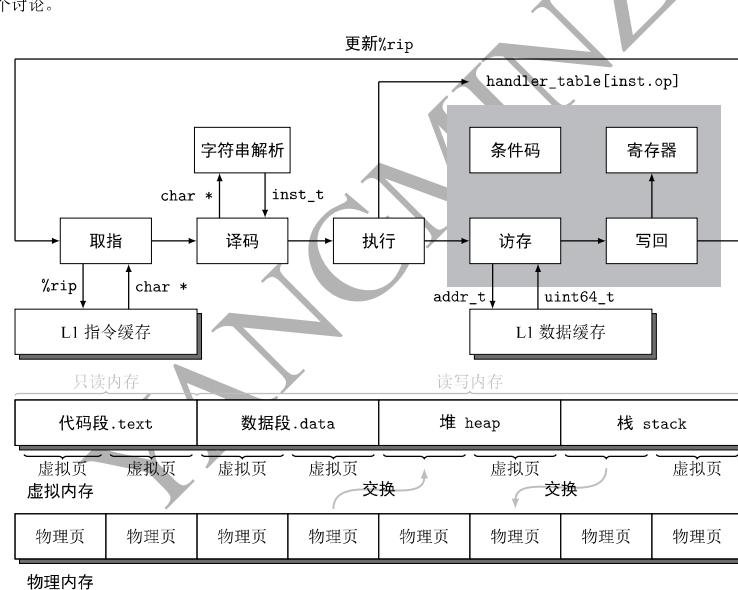


图 4: CPU 指令周期近似

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>

计算机系统

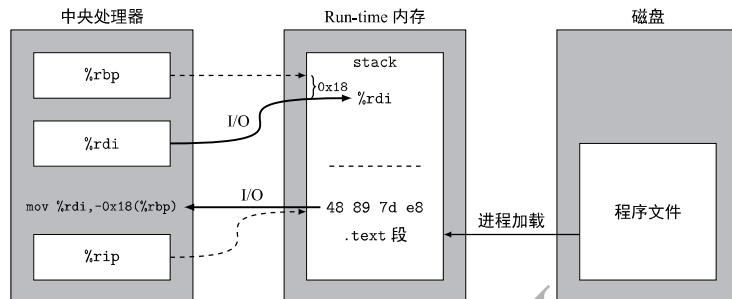


图 5: 指令处理

1.2 汇编指令的语法分析

大致了解计算机指令的硬件资源以及原理以后，我们便开始建立指令了。这时，最迫切的问题是，指令是怎样储存在内存上的，以及它的格式是怎样的。对此，我们利用字符串做出统一的回答。

通常，计算机系统中 CPU 的指令是按照 Byte 为单位储存的，在读取、翻译、执行时，也是按照 Byte 单位对指令进行翻译。这就是机器代码（Machine-Level Code）。在我们对计算机系统的近似中，我们不采用 Byte 为单位的指令格式，而将其看作一种和 C 语言一样的语言，进而解析与翻译。例如，机器码“0x48 0x89 0xd3”在指令集中被翻译为“`mov %rdx,%rbx`”，而我们直接用字符串“`mov %rdx,%rbx`”来表示这一段汇编指令。

这样做的好处是字符串指令集变得可移植，作为对机器码 Byte 的一层抽象，字符串的汇编指令类似于虚拟机或中间语言，同时方便我们从文件中读取写入（特别是在处理链接的重定位时）。不仅如此，我们在模拟中甚至可以指定字符串所占用的长度，例如定义每一个字符串都是 `char[64]`，多余的 `char` 可以用 '`\0`' 去填充。这样，我们就实现了一个在内存占用上定长的指令，并且 `strlen()` 仍能读取该字符串的真实长度。

为了翻译字符串汇编指令，我们需要对它进行语法分析（Syntax Analysis）。在这一步，我们只是解析字符串的格式，并不对指令本身做任何计算。

汇编指令的操作符（Operator）主要有三种类型：(1) 无操作数的操作符，例如 `nop`, `ret`, `leave`; (2) 单操作数的操作符，例如 `call`, `push`, `pop`, `jmp`; (3) 双操作数的操作符，例如最常见的 `mov`, `add`, `cmp`。其中，对于双操作数，我们指定其格式为 `op src,dst`，即左操作数为源操作数，右操作数为目的操作数。我们以不同操作数数量的操作符为终止归约的最小终结符（Terminal）或标志符（Token），可以建立最顶层的语法（Grammar）：

$$E \rightarrow Op \mid Op\ F \mid Op\ F,F$$

在上述语法中，非终结符（Nonterminal） F 可以向下展开，进一步描述操作数（Operand）。操作数主要分为十一种格式，三种主要的类型：立即数（Immediate number），寄存器（Register），内存地址（Memory Address）。其中内存地址有九种格式，可以进一步被向下展开。因此，我们首先建立对三种类型的语法：

$$F \rightarrow \$\text{Imm} \mid \% \text{Reg} \mid \text{Imm} \mid (\text{M}) \mid \text{Imm}(\text{M})$$

在上述语法中，最后三种是有关内存地址的。立即数与寄存器均为终结符，内存地址格式可以继续向下展开，最终全部被展开为终结符：

$$M \rightarrow \% \text{Reg} \mid \% \text{Reg}, \% \text{Reg} \mid , \% \text{Reg}, \text{Scale} \mid \% \text{Reg}, \% \text{Reg}, \text{Scale}$$

需要我们注意的是，上述语法并不能直接被用于语法分析，只是我们对汇编指令格式最直观的描述。为了实现语法分析，我们要保证每次向下展开时，被展开的各个子项的第一个可能的字符集（也即 FIRST 集）互不相交。举一个例子，对于指令“mov %rsp,%rbp”，编译器在解析到第一个‘%’时，它无法确定应当被 $E \rightarrow F$ 或是 $E \rightarrow F, F$ 展开。

因此，右侧被展开每一个语句应当从不同的字符开始。而右侧任一语句的所有可能的第一个字符，也就构成了 FIRST 集（First Set）。这类似于一个 $\text{char} \rightarrow (\text{void} *)$ 的映射，根据第一个字符去查找展开的规则。这样，我们必须要求右侧各个语句的 FIRST 集合互不相交。为了改写语法，我们需要增加一个空字符串（Empty String），并且用 ϵ 表示：

$$\begin{aligned}
 1. \quad E &\rightarrow \text{Op} \ F \\
 2. \quad F &\rightarrow \epsilon \mid GH \\
 3. \quad H &\rightarrow \epsilon \mid G \\
 4. \quad G &\rightarrow \$\text{Imm} \mid \% \text{Reg} \mid \text{Imm}T \mid (\text{M}) \\
 5. \quad T &\rightarrow \epsilon \mid (\text{M}) \\
 6. \quad M &\rightarrow \% \text{Reg}P \mid , \% \text{Reg}, \text{Scale} \\
 7. \quad P &\rightarrow \epsilon \mid \% \text{Reg}Q \\
 8. \quad Q &\rightarrow \epsilon \mid , \text{Scale}
 \end{aligned}$$

接下来，我们对上述文法计算 FIRST 集。FIRST 集的计算有三条规则，我们不必一一详述，只需要知道 FIRST 集的含义是被展开的所有可能的第一个字符的集合即可，因此左侧表达式的 FIRST 集一定是从右侧 FIRST 集计算得到的。为了保证映射 $\text{char} \rightarrow (\text{void} *)$ 成立，要求右侧所有被展开项的 FIRST 集互不相交，因此我们自下而上从终结符向上到 E 计算：

$$\begin{aligned}
 \text{FIRST}(Q) &= \{\epsilon, ',', '\}\} \\
 \text{FIRST}(P) &= \{\epsilon, '%''\} \\
 \text{FIRST}(M) &= \{'%', ',', '\}\} \\
 \text{FIRST}(T) &= \{\epsilon, '('\} \\
 \text{FIRST}(G) &= \{'$', '%', '0'-'9', '-', '('\} \\
 \text{FIRST}(H) &= \{\epsilon, ','\} \\
 \text{FIRST}(F) &= \{\epsilon, '$', '%', '0'-'9', '-', '('\} \\
 \text{FIRST}(E) &= \text{FIRST}(\text{Op})
 \end{aligned}$$

除了 FIRST 集外，为了确定 $\text{char} \rightarrow (\text{void} *)$ 映射，在 FIRST 集取空字符 ϵ 的情况下，我们还需要向后观察字符，才能确定用于展开的规则，如图6：

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>
计算机系统



图 6: FIRST 集与 FOLLOW 集

因此我们需要构造非终结符后继的**FOLLOW**集: 对任意非终结符 A , 其 FOLLOW 集 $FOLLOW(A)$ 是所有可能出现在它右侧的第一个字符的集合。对于 FOLLOW 集, 我们不必再考虑空字符 ϵ 。对于形如 $A \rightarrow \alpha B \beta$ 的表达式, 显然 $FOLLOW(B)$ 中包括非空字符的 $FIRST(\beta)$ 元素。若有 $\epsilon \in FIRST(\beta)$ 或直接有 $A \rightarrow \alpha B$, 则 $FOLLOW(B)$ 中有 $FOLLOW(A)$ 的元素: $A\gamma \rightarrow \alpha B\gamma$ 。这样, 我们自上而下到终结符计算 FOLLOW 集, 显然 E 的后继就是字符串的终止符 '\0':

$$\begin{aligned}
 FOLLOW(E) &= \{\backslash 0\} \\
 FOLLOW(F) &= \{\backslash 0\} \\
 FOLLOW(G) &= \{\backslash 0, ',', '\} \\
 FOLLOW(H) &= \{\backslash 0\} \\
 FOLLOW(T) &= \{\backslash 0, ',', '\} \\
 FOLLOW(M) &= \{)\} \\
 FOLLOW(P) &= \{)\} \\
 FOLLOW(Q) &= \{)\}
 \end{aligned}$$

接下来, 利用 FIRST 集与 FOLLOW 集, 我们建立对语句的预测表 (Parsing Table), 对每一个表达式 A , 当接收某一终结符 t 时, Parsing Table 决定采用某一种规则 $A \rightarrow \alpha$ 去进行归约。从我们已经计算的 FIRST 集与 FOLLOW 集看, 对于 $A \rightarrow \alpha$, $FIRST(\alpha)$ 中所有非 ϵ 字符都对应了特定的规则, 这也就是我们先前所说的 $\text{char} \rightarrow (\text{void} *)$ 映射。如果 $FIRST(\alpha)$ 中含有空字符 ϵ , 则考虑 FOLLOW 集, 用来检测是否忽略这个表达式。基于这个规则, 我们建立预测表。

	Op	'\$'	'%'	'.'	Num	'('	')'	'\0'
$E \rightarrow$	$\text{Op } F$							
$F \rightarrow$		GH	GH		GH	GH		ϵ
$G \rightarrow$		$\$Imm$	$\%Reg$		$ImmT$	(M)		
$H \rightarrow$				$,G$				ϵ
$T \rightarrow$				ϵ		(M)		ϵ
$M \rightarrow$			$\%Reg^P$	$, \%Reg, Scale$				
$P \rightarrow$			$\%Reg^Q$				ϵ	
$Q \rightarrow$					$Scale$		ϵ	

建立 Parsing Table 以后, 我们可以用它进行非递归的预测分析 (Nonrecursive Predictive Parsing)。例如解析指令 "mov \$0xff, (%rax)":

已匹配	归约栈	表达式	操作
	E	<code>mov \$0xff,%rax</code>	
$\text{Op } F$	$\text{Op } F$	<code>mov \$0xff,%rax</code>	展开 $E \rightarrow \text{Op } F$
$\text{mov } F$	$\text{mov } F$	<code>\$0xff,%rax</code>	匹配 <code>mov</code>
<code>mov</code>	GH	<code>\$0xff,%rax</code>	展开 $F \rightarrow GH$
<code>mov</code>	$\text{Imm}TH$	<code>\$0xff,%rax</code>	展开 $G \rightarrow \text{Imm}T$
<code>mov</code>	$\$0\xff TH$	<code>,%rax</code>	匹配 <code>\$0xff</code>
<code>mov \$0xff</code>	H	<code>,%rax</code>	展开 $T \rightarrow \epsilon$
<code>mov \$0xff</code>	$,G$	<code>%rax</code>	展开 $H \rightarrow ,G$
<code>mov \$0xff</code>	$,G$	<code>%rax</code>	匹配 $,$
<code>mov \$0xff,</code>	(M)	<code>%rax</code>	展开 $G \rightarrow (M)$
<code>mov \$0xff,</code>	(Reg)	<code>%rax</code>	展开 $M \rightarrow \%Reg$
<code>mov \$0xff,</code>	$(\%rax)$	<code>%rax</code>	匹配 $(\%rax)$
<code>mov \$0xff,(\%rax)</code>			

到此为止，我们可以在线性时间内解析字符串指令代码了。在这个过程中，我们建立了抽象语法树（Abstract Syntax Tree），如图7所示。

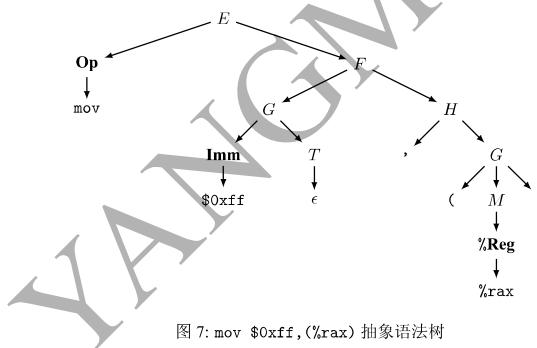


图 7: `mov $0xff,(%rax)` 抽象语法树

在实际实现中，由于汇编指令的格式非常简单，我们未必需要经过上述复杂的语法分析。更加简便的做法是计算标志性的符号，将字符串分割为若干部分，分别处理，如图8。对于汇编代码，我们可以对空格' '，左右括号'('，')' 以及逗号',' 进行计数。显然，Count['('] == 0 且 Count[','] == 0 时，字符串处在 **Op** 段。操作符和两个操作数在字符串上的范围都可以通过类似的计数方法判断出来。

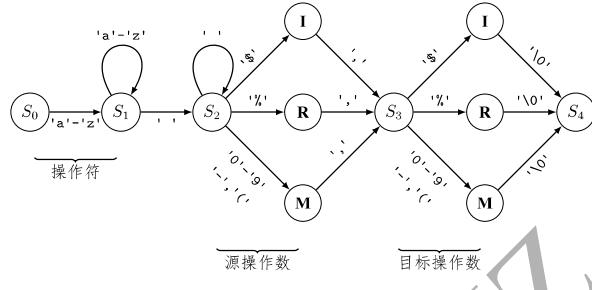


图 8: 扫描与计数解析

1.3 汇编指令的词法分析

在上述语法分析中，我们忽略了如何匹配终结符，现在我们要讨论这一部分，匹配终结符的过程即词法分析 (Lexical Analysis)。

对于汇编指令的终结符，主要分为三种类型：数字的立即数 (Imm) 与变址比例因子 (Scale)，字符的寄存器 (Reg) 和操作数 (Op)，以及其他符号，包括空格 (' ')、逗号 (',')、左右圆括号 ('(', ')')。对于数字类的终结符，我们可以采用确定有限自动机 (Deterministic Finite Automaton, DFA)，按照正则文法 (Regular Grammar) 去判断与解析，见图9。需要注意的是，汇编指令中的数字包括十六进制和十进制的正负整数，同时我们需要在每一步状态转移时检查数值是否溢出。

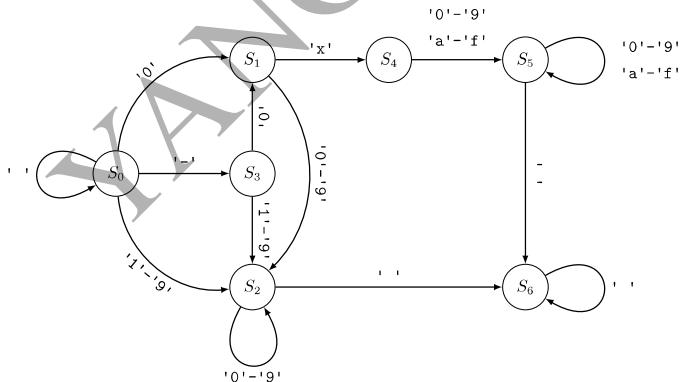


图 9: 有限自动机计算十进制与十六进制数

对于字符类型的终结符，即操作数与寄存器，我们可以采取简单的字符串匹配 (Matching)。在这里，我们有多种数据结构可以采用。最通常的做法是采用数组或列表，存储所有可能的字符串值，例如：{"rax", "rbx", ...} 及 {"push", "pop", ...}，遍历这个数组并对每一个元素进行匹配。采用这样的数据结构，

则查找与匹配的时间复杂度为 $O(L \cdot E[\text{strlen}])$ ，其中 $E[\text{strlen}]$ 是字符串的平均长度。另一种策略是对上述的字符串数组建立字典树（Trie）或前缀树（Prefix Tree），图10就是 `rax` 系列的寄存器的前缀树。

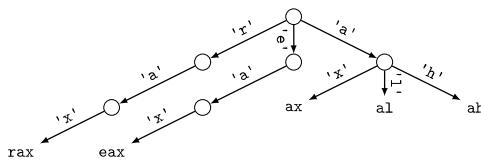


图 10: %rax 寄存器前缀树

利用前缀树，搜索和匹配便可以不必遍历整个数组，而时间复杂度减小到 $O(E[\text{strlen}])$ 。在实现中，可以在执行 CPU 指令周期前对所有的指令操作符名、寄存器名建立前缀树，这样在执行指令周期时，就可以减小匹配时间。前缀树的每个节点通常包括一个 $\text{char} \rightarrow (\text{void}^*)$ 映射的数据结构，其中 char 为字符， (void^*) 为子树的地址。可以看到，前缀树与 DFA 很相似，所不同的是前缀树自动建立了状态转移（因此它不够精简），而 DFA 需要我们预先计算好状态转移。

经历过语法分析与词法分析以后，我们可以将一个字符串的指令映射到指令的结构中，并且确定操作符、操作数的类型：

```
/*src/headers/instruction.h

1 typedef struct OPERAND_STRUCT
2 {
3     od_type_t    type;      // IMM, REG, MEM
4     uint64_t      imm;       // immediate number
5     uint64_t      scal;      // scale number to register 2
6     uint64_t      reg1;      // main register
7     uint64_t      reg2;      // register 2
8 } od_t;
9
10 typedef struct INST_STRUCT
11 {
12     op_t          op;        // enum of operators. e.g. mov, call, etc.
13     od_t          src;       // operand src of instruction
14     od_t          dst;       // operand dst of instruction
15 } inst_t;
```

其中 `op_t` 和 `od_type_t` 都是 C 语言中的枚举类型 (`enum`)。其中 `op_t` 描述了我们所近似的全部指令类型 (操作符类型), 例如 `INST_PUSH`, `INST_POP` 等。`od_type_t` 则描述全部操作数: 立即数 (`IMM`), 寄存器 (`REG`), 以及九种内存格式 (`MEM_IMM`, `MEM_REG1`, `MEM_IMM_REG1` 等)。举一个例子, 我们观察对数组指针 `uint64_t *ptr` 中的元素的引用, 如图11:

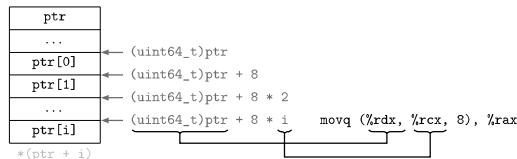


图 11: 引用数组元素 `ptr[i]` 的指令

最后, 我们根据操作数的类型, 将操作数计算为一个 `uint64_t` 的值。这个值可以是立即数, 可以是 CPU 中近似寄存器的虚拟地址, 可以是虚拟内存的地址, 完全根据操作数的类型进行计算:

```
/src/hardware/cpu/isa.c

1 static void      parse_instruction  (const char *str, inst_t *inst);
2 static void      parse_operand     (const char *str, od_t *od);
3 static uint64_t  compute_operand  (od_t *od);
```

到此为止, 我们已经将一个 64 字符的字符串映射到了一个指令数据结构了。这部分工作可以被视为译码 (Decode)。接下来, 我们需要做的就是根据指令的类型, 以及操作数的数值, 去进行计算。计算的过程, 考虑到我们对“计算”的理解, 其实就是更新 CPU 和内存的状态。

1.4 CPU 指令周期

正如我们在解释“指针”时所说的, 以及根据 Von Neumann 对指令的理解, 函数与指令都是数据, 因此可以通过指针访问。因此, 我们可以将函数代码的起始地址 (指针) 保存起来, 存放在一个查找表中, 这就是函数指针数组。这种回调机制将函数注册在数组中, 当条件触发时通过数组调用函数。

我们首先申明各种 CPU 所将执行的指令, 用作回调。例如:

```
/src/hardware/cpu/isa.c

1 static void push_handler          (od_t *src_od, od_t *dst_od);
2 static void pop_handler           (od_t *src_od, od_t *dst_od);
3
4 // handler table storing the handlers to different instruction types
5 typedef void (*handler_t)(od_t *, od_t *);
6
7 // look-up table of pointers to function
8 static handler_t handler_table[NUM_INSTRTYPE] = {
9     // ...
10}
```

```
10     &push_handler,           // 1
11     &pop_handler,           // 2
12     // ...
13 };
```

这样，我们就可以通过指令结构 `inst_t` 中的 `op` 域去方便地调用指令（因为枚举类型在 C 语言中可以被解释为整数类型）。基于上述的函数指针数组，我们便可以近似地实现图4中所描述的指令周期了：

```
/src/hardware/cpu/isa.c

1 // instruction cycle is implemented in CPU
2 // the only exposed interface outside CPU
3 void instruction_cycle()
4 {
5     // FETCH
6     char inst_str[MAX_INSTRUCTION_CHAR + 8];
7     readinst_dram(va2pa(cpu_pc.rip), inst_str);
8
9     // DECODE
10    inst_t inst;
11    parse_instruction(inst_str, &inst);
12
13    // EXECUTE
14    handler_t handler = handler_table[inst.op];
15    handler(&(inst.src), &(inst.dst));
16 }
```

在这一节，我们先考虑部分递归函数中三个基本函数的实现。与它们最为相关的，是 `mov`, `add` 等指令。

置零函数

置零函数在 URM 中是将结果贮存器置零，在常见的计算机系统中，保存结果的寄存器通常是`%rax`，因此我们考虑对`%rax`的置零即可。更进一步，我们考虑对任意寄存器以及任意内存，置类型为任意 `uint64_t` 的值，这些操作通过 `mov` 置零即可实现。

在真实的 X86 指令集中，`mov` 指令对不同长度的数据类型分为若干条指令：`movb` 移动 8 位，`movw` 移动 16 位，`movl` 移动 32 位，`movq` 移动 64 位。正如我们先前所说的，对于 C 语言而言，并不存在数据类型的差别。数据类型被视作不同长度的二进制数据，因此在生成机器指令时，C 语言中数据类型的抽象已经被展开了。

在我们的实现中，我们并不考虑不同长度的数据类型，而通用 `uint64_t` 表达二进制串。将二进制串解释为不同的数据类型，这是指令的责任。对于 `mov` 指令，我们考虑不同的操作数：立即数、寄存器、虚拟内存。按照先前所描述的函数指针，我们实现有关 `mov` 的函数如下：

```
/src/hardware/cpu/isa.c

1 static void mov_handler(od_t *src_od, od_t *dst_od)
2 {
```

```
3     uint64_t src = compute_operand(src_od);
4     uint64_t dst = compute_operand(dst_od);
5
6     if (src_od->type == REG && dst_od->type == REG)
7     {
8         // src: register
9         // dst: register
10        *(uint64_t *)dst = *(uint64_t *)src;
11        cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
12        cpu_flags._flags_value = 0;
13        return;
14    }
15    else if (src_od->type == IMM && dst_od->type >= MEM_IMM)
16    {
17        // src: immediate number
18        // dst: virtual address
19        write64bits_dram(va2pa(dst), src);
20        cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
21        cpu_flags._flags_value = 0;
22        return;
23    }
24    // ...
25 }
```

后继函数

后继函数是对结果寄存器加一，更进一步的，我们可以对结果寄存器加任意数值，也即 add 指令。add 指令同样需要考虑数据类型的长度。除此以外，add 指令还需要考虑两个方面：负数的加法以及设置 CPU 条件码。

由于我们通用 `uint64_t` 表达二进制串，也即一切数据类型，因此也需要考虑负数的加法。实际上，由于 2 的补码机制本质上是取余，因此对补码形式的负数，加法运算同样成立，只是要通过溢出的形式实现。例如我们在 3-Bit 的情形下做加法： $(-1) + (-2) = 111 + 110 = 1101$ ，保留低三位到 101，并不发生有符号数的溢出，但发生无符号数的溢出。CPU 条件码的置位规则如前所述。这样，add 指令实现为：

```
/src/hardware/cpu/isa.c
1 static void add_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t src = compute_operand(src_od);
4     uint64_t dst = compute_operand(dst_od);
5
6     if (src_od->type == REG && dst_od->type == REG)
7     {
8         // src: register
9         // dst: register
```

```
10     uint64_t val = *(uint64_t *)dst + *(uint64_t *)src;
11
12     uint64_t sign_src = (((*uint64_t *)src) >> 63) & 0x1;
13     uint64_t sign_dst = (((*uint64_t *)dst) >> 63) & 0x1;
14     uint64_t sign_val = (val >> 63) & 0x1;
15     cpu_flags.CF = (val < *(uint64_t *)src);
16     cpu_flags.ZF = (val == 0x0);
17     cpu_flags.SF = sign_val;
18     cpu_flags.OF = (!(sign_src ^ sign_dst) && (sign_val ^ sign_src));
19
20     *(uint64_t *)dst = val;
21     cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
22     return;
23 }
24 }
```

投影函数

最后，投影函数仍然可以用 `mov` 指令实现，我们就不再描述了。但作为这一节的结尾，我们可以简单描述一下 `mov` 和 `add` 所使用的硬件资源，使我们对理论计算机的实现有更加底层的理解。

首先考虑 `mov` 指令，如果两个操作数都是寄存器，那么 CPU 可以直接将源寄存器中储存的值拷贝到目标寄存器。如果操作数中包括内存地址，那么 CPU 要进行内存访问（可能直接命中缓存）。其实，在不考虑硬件效率和同步的情况下，我们完全可以将 CPU 内的寄存器看作内存，或将内存看作寄存器，它们在 URM 中都扁平地被抽象为贮存器。在置零函数中，`mov` 指令将立即数 `0x0` 拷贝到贮存器；在替换函数中，`mov` 指令将贮存器（寄存器或内存）上的值拷贝到 `%rax` (R_1)。

`add` 指令实现了后继函数。URM 可以通过执行 n 次后继指令实现 $+n$ ，而 `add` 指令将源寄存器和目标寄存器中的值传到 CPU 中的算术逻辑单元（Arithmetic and Logic Unit, ALU），通过 ALU 直接计算出结果，并且写回到目标寄存器。`add` 指令还扩展了对负数的运算，并且设置 CPU 的条件码。

考虑过三种基本函数的物理实现后，我们继续考虑部分递归函数的生成方法，也即三种算子： σ, ρ, μ 。这样，我们就实现了对一个简陋的物理计算机的近似了。

过程调用

σ 算子其实也就是 C 语言中的过程调用（Procedure Call）或函数调用。为了更加清晰地理解相关的汇编指令（`push`, `pop`, `call`, `ret` 等），我们在这里从两条路线去理解。第一条路线是部分递归函数的，也即复合函数 $h(g_1, \dots, g_m)$ 。显然，从逻辑的角度看，函数的复合过程（ σ 算子）与 C 语言中调用函数是相同的。抛开一切底层的实现，C 语言关于函数调用的语法实际上就是复合函数。第二条路线是 URM 的，数学上的理想计算机与物理计算机的实现有相似之处，因此相关 `x86` 指令其实就是 URM 组合程序的过程。在这两条路线下，我们其实不严谨地完成了 \mathcal{P} 与 \mathcal{R} 等价性的部分证明。

我们首先考虑一个简单的问题，由 Z, S, T, J 指令构成的序列 P 如何计算一个函数 $f : \mathbb{N}_0^n \mapsto \mathbb{N}_0$ 。在 URM 上，我们这样设置：贮存器 R_1 为只储存程序 P 的最后计算结果，也即 $f(\mathbf{x})$ ；贮存器 R_2, \dots, R_{n+1} 储存程序 P 所需的参数，也即 $\mathbf{x} = [x_1, \dots, x_n]$ ；最后，从 R_{n+2} 开始，贮存器被用作中间计算。我们认为函数 f 是可计算的，程序 P 是可停机的，因此指令的数量是有限的，所需的贮存器数量也是有限的。这样，假设程序 P 至多所需要的贮存器序号为 $m(P)$ 。当计算结束后， R_1 被写回计算结果，如图12。

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>
计算机系统

$$\begin{array}{c}
 f(x) \\
 \overbrace{\quad\quad\quad}^{R_1} \quad \overbrace{\quad\quad\quad}^{R_2 \quad R_3 \quad \cdots \quad R_n \quad R_{n+1}} \quad \overbrace{\quad\quad\quad}^{R_{n+2} \quad R_{n+3} \quad \cdots \quad R_{m(P)}}
 \end{array}$$

图 12: URM 程序与可计算函数的转换

上述函数 f 与程序 P 在 C 语言中与汇编指令中可以这样理解：对于 C 语言，每一个程序只有一个源文件，源文件中只有一个 `main()` 函数；对于汇编指令，在虚拟内存上，`main()` 函数所对应的汇编代码从 `0x00400000` 开始。这样就是形式上最简单的 C 语言程序与汇编指令，也是 C 语言的初学者最经常动手写出来的程序。这一形式隐含了对程序指令 $P = I_1, I_2, \dots, I_s$ 的一个要求： $(\forall 1 \leq t \leq s)(I_t = J(m, n, q)) \rightarrow (q \leq s)$ ，也即程序 P 的任意指令 I_t 不可以跳转离开 P 。满足这一条件的 URM 程序被称为标准形式（Standard Form）的程序。

接下来，我们考虑最简单的复合函数，全部采用单变量：内层函数 $g : \mathbb{N}_0 \mapsto \mathbb{N}_0$ 以及外层函数 $h : \mathbb{N}_0 \mapsto \mathbb{N}_0$ ，并且它们所对应的 URM 程序 P_g 与 P_h 都是标准形式的程序。考虑 P_g 与 P_h 最大所用的贮存器序号 $u = \max(m(P_g), m(P_h))$ ，则在 $[1, u]$ 区间内的贮存器用作计算，然后将参数 x 存放在 R_{u+1} ，将内层函数计算结果 $g(x)$ 存放在 R_{u+2} 。这些拷贝数据的工作只需要有限次调用 T 指令即可。因此，如果 P_g 与 P_h 可停机，则 $P_{h(g)}$ 可停机。

URM 与复合函数与 C 语言的过程调用是相似的：所有被用来计算的贮存器上的数据，最终都被归约到结果贮存器 R_1 之中。但是，复合函数 $h(g_1, \dots, g_k)$ 的计算是单向的，而 C 语言的函数调用要求内层函数 g_i 计算结束后，计算机能重新返回外层函数 h ，因为外层函数 h 在取得 g_{i+1} 以前可能也会进行计算。

因此，我们不能直接覆盖计算部分的贮存器，而需要保存 h 的计算现场。可重进入的要求使得计算顺序不再是一个单向的链表： $g_1 \rightarrow g_2 \rightarrow \dots \rightarrow g_k \rightarrow h$ ，而天然地形成了一个树结构： $h \rightarrow g_1 \rightarrow h \rightarrow g_2 \rightarrow h \rightarrow \dots \rightarrow g_k \rightarrow h$ 。这样，我们就需要将贮存器组织成一个先进后出的栈（Stack），用来保存 C 语言函数所计算的数据。

我们可以利用 λ 演算的语法进一步理解。我们知道，树是一种递归的定义：如果 T 是树，那么它的子节点也都是树。函数与程序也是一样的，考虑 λ 表达式：如果 E 是一个 λ 表达式，那么根据语法展开 $E \rightarrow (EE)$ ，在右侧的子项也分别是 λ 表达式。同样的，函数 h 所调用的函数 g_1, \dots, g_k 也都是函数。

在先前的描述中，一段 URM 程序或可计算的递归函数，在计算中会用到有限的贮存器空间。在 C 语言程序运行时，我们称这部分内存为函数的帧（Frame），它是被保存在栈上的数据结构。与 URM 程序一样，我们主要需要考虑的是传入的参数与返回的结果。除此以外，由于使用了栈结构，我们还需要额外考虑描述栈帧的状态。

<https://github.com/yangminz>
<https://space.bilibili.com/4564101>
<https://www.zhihu.com/people/zhao-yang-min>

计算机系统

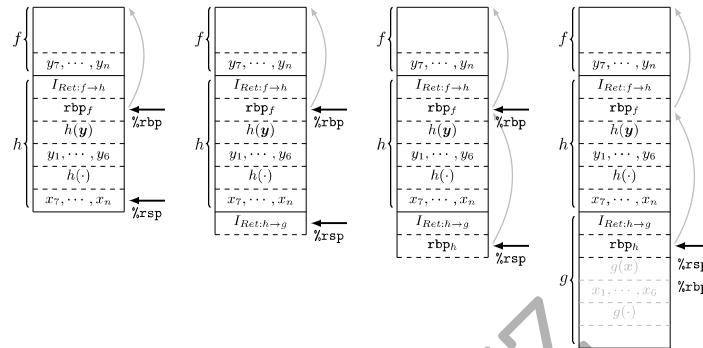


图 13: call, push, mov 指令

如图13，过程 g 的参数 x 被分为两部分。当 h 调用 $g(x_1, \dots, x_n)$ 之前，前六个参数依次分别被寄存器 $\%rdi, \%rsi, \%rdx, \%rcx, \%r8, \%r9$ 保存，余下的参数依次被写入内存。在 h 的栈帧内，CPU 通过 push 指令向低地址增长栈指针，扩张栈的大小，同时将调用 g 所需的参数 x_7, \dots, x_n 写入内存。这样做是因为 CPU 的寄存器资源非常珍贵，而内存的资源相对充足，所以需要内存去辅助寄存器进行参数传递（Parameter Passing）。

```

push  xn
...
push  x7
mov   x6,%r9
mov   x5,%r8
mov   x4,%rcx
mov   x3,%rdx
mov   x2,%rsi
mov   x1,%rdi

```

调用 g 需要执行 call 指令，call 指令将当前的返回地址压入栈内，也即将 $\%rsp$ 减去一个指针的大小（64 位机器中为一个 `uint64_t`），同时将返回地址写入 $\%rsp$ 所指向的内存。假定 $I_t = \text{call}$ ，那么返回地址（Return Address）就是指令 I_{t+1} 的地址。最后，更新 $\%rip$ 进行跳转：

```

/src/hardware/cpu/isa.c

1 static void call_handler(od_t *src_od, od_t *dst_od)
2 {
3     // src: procedure instruction address
4     uint64_t src = decode_operand(src_od);
5
6     cpu_reg.rsp = cpu_reg.rsp - 8;
7     write64bits_dram(va2pa(cpu_reg.rsp),
8         cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR);
9     cpu_pc.rip = src;

```

```
10     cpu_flags._flags_value = 0;
11 }
```

到此为止，程序计数器已经跳转到过程 *g* 的起始地址。为了还原函数调用的现场，被调函数 *g* 需要保存调用函数 *h* 的栈帧信息。`%rsp` 向下压入`%rbp`的数值，方便将来还原 *g*。为此，我们实现 `push` 指令，它的实现也就是将`%rsp` 向内存低地址增长 `uint64_t`，然后将操作数的值写入该地址：

```
/src/hardware/cpu/isa.c
1 static void push_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t src = decode_operand(src_od);
4     if (src_od->type == REG)
5     {
6         // src: register
7         // dst: empty
8         cpu_reg.rsp = cpu_reg.rsp - 8;
9         write64bits_dram(va2pa(cpu_reg.rsp), *(uint64_t *)src);
10        cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
11        cpu_flags._flags_value = 0;
12        return;
13    }
14 }
```

在保存过`%rbp`的数值以后，我们便进入被调函数 *g* 的栈帧。通常，我们直接将`%rsp`的值赋给`%rsp`，这便完全进入了 *g* 的栈帧。在这以后，我们用 `mov` 指令，将保存在寄存器`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` 上的参数 x_1, \dots, x_6 的值拷贝到 *g* 的栈帧：

```
mov    %rdi,-0x18(%rbp)
mov    %rsi,-0x20(%rbp)
mov    %rdx,-0x28(%rbp)
mov    %rcx,-0x30(%rbp)
mov    %r8,-0x38(%rbp)
mov    %r9,-0x40(%rbp)
```

在参数以后的栈帧内存被用作计算 *g()*，计算的结果，也就是 C 语言函数的返回值，通常被保存在寄存器`%rax` 以内，例如直接从 ALU 中计算到`%rax`。但返回值同样要被保存到内存上（在`%rbp`所指的位置之下），再被读到`%rax`，交给被调函数 *h* 使用：

```
mov    %rax,-0x8(%rbp)
mov    -0x8(%rbp),%rax
```

当计算完成以后，我们需要恢复函数 *h* 的现场，最主要的就是恢复帧指针`%rbp`的数值，使它指向 *h* 的栈帧地址。

```
/src/hardware/cpu/isa.c
1 static void pop_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t src = decode_operand(src_od);
4     if (src_od->type == REG)
```

```
5     {
6         // src: register
7         uint64_t old_val = read64bits_dram(va2pa(cpu_reg.rsp));
8         cpu_reg.rsp = cpu_reg.rsp + 8;
9         *(uint64_t *)src = old_val;
10        cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
11        cpu_flags._flags_value = 0;
12        return;
13    }
14 }
```

最后，将程序计数器恢复到返回地址，继续执行 h 的指令。

```
/src/hardware/cpu/isa.c
1 static void ret_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t ret_addr = read64bits_dram(va2pa(cpu_reg.rsp));
4     cpu_reg.rsp = cpu_reg.rsp + 8;
5     cpu_pc.rip = ret_addr;
6     cpu_flags._flags_value = 0;
7 }
```

递归与循环

接下来，我们考虑基本递归。实际上，利用上一节所描述的 C 语言过程控制，我们本质上已经实现了递归，也即一个函数调用它的自身。这是通过指向函数本身的指针，利用函数栈而实现的，也就是基于 Von Neumann 的思想。除了递归的过程以外，我们只需要考虑递归的停止条件，这就需要我们实现对程序计数器 rip 的跳转，也即 jmp 一系列的指令，我们用正则表达式简记为 j*。

j* 指令的实现非常直接。在 j* 指令之前，我们设置 CPU 中的条件码 CF, ZF, SF, OF。我们以比较两个操作数的指令 cmp 为例，cmp 指令本质上是通过减法运算去设置条件码，只是不将减法的结果写回寄存器：

```
/src/hardware/cpu/isa.c
1 static void cmp_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t src = compute_operand(src_od);
4     uint64_t dst = compute_operand(dst_od);
5
6     if (src_od->type == REG && dst_od->type == REG)
7     {
8         // src: register
9         // dst: register
10        src = 0x1 + -(*(uint64_t *)src);
11        dst = *(uint64_t *)dst;
12        uint64_t val = dst + src;
13
14        uint64_t sign_src = (src >> 63) & 0x1;
15        uint64_t sign_dst = (dst >> 63) & 0x1;
16        uint64_t sign_val = (val >> 63) & 0x1;
```

```

17     cpu_flags.CF = dst < val;
18     cpu_flags.ZF = (val == 0x0);
19     cpu_flags.SF = sign_val;
20     cpu_flags.OF = ((sign_dst ^ sign_src) && !(sign_src ^ sign_val));
21
22     cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
23     return;
24 }
25 }
```

在 CPU 条件码的基础上，`j*` 指令只需要根据条件码来实现即可。例如我们实现 `jne` 指令，要求在上一个指令的两个操作数不相等的条件下跳转：

```

/src/hardware/cpu/isa.c

1 static void jne_handler(od_t *src_od, od_t *dst_od)
2 {
3     uint64_t src = compute_operand(src_od);
4     // src: immediate number of the target jumping address
5     if (cpu_flags.ZF == 0x1)
6     {
7         cpu_pc.rip = cpu_pc.rip + sizeof(char) * MAX_INSTRUCTION_CHAR;
8     }
9     else
10    {
11         cpu_pc.rip = src;
12     }
13     cpu_flags._flags_value = 0;
14 }
```

利用 `j*` 指令，我们就可以对程序进行条件判断，进而确定递归的停止条件，也即是内存栈向下停止增长的条件。从硬件和体系结构的实现看，`j*` 指令的跳转是可以优化的。我们可以对条件分支进行预测，但这个话题不在我们的讨论范围之内了。

有了条件判断 `cmp` 与条件跳转 `j*`，我们便实现了 URM 基本指令中的 $T(m, n, q)$ 。接下来，我们就可实现递归与循环了。

实际上，我们在 URM 和递归函数中都没有定义有关循环（Loop）的操作。在 URM 的指令中，循环是容易实现的；但在递归函数中循环并不那么显然。直到现在，我们才可以通过递归操作去实现函数的循环。以计算 Fibonacci 数列的第 n 项 $F_n = F_{n-1} + F_{n-2}$ 为例，我们通常有三种算法去实现。第一种算法将递推关系扩写： $F_n = F_{n-1} + F_{n-2}$ 以及 $F_{n-1} = F_{n-1} + 0$ 。这样，就可以利用矩阵表达递推关系：

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = M \cdot \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = M^{n-1} \cdot \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = M^{n-1} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

从 M^1 开始直接计算矩阵幂次 M^{n-1} ，算法复杂度是 $O(n)$ ，但我们可以通过对分治法去计算快速幂： $M^{n-1} = M^{\frac{n-1}{2}} \cdot M^{\frac{n-1}{2}}$ ，其中 $M^{\frac{n-1}{2}}$ 只需要被计算一次。假定计算 F_n 的时间复杂度是 $T(n)$ ，由于 $M^{\frac{n-1}{2}}$ 只被计算一次，因此 $T(n) = T(\frac{n}{2}) + O(1)$ 。这样，算法复杂度被优化到 $T(n) = O(\log_2(n))$ 。

第二种算法是有循环的，我们从第 0 项开始向后计算，一直计算到第 n 项为止，也即 $T(n) = T(n-1) + O(1)$ ，这样，算法的复杂度是线性的： $T(n) = O(n)$ 。并且由于没有重复计算，因此循环的算法无法优化。

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \dots, F_n$$

```

1 int Fibonacci(int n)
2 {
3     int a = 0, b = 1, c = -1, i = 2;
4 Loop:
5     c = a + b;           // 6ad <Fibonacci+0x23>
6     a = b; b = c;
7     i = i + 1;
8     if (i < n) {        // cmp -0x14(%rbp),%eax
9         goto Loop;      // j1 6ad <Fibonacci+0x23>
10    }
11    return c;
12 }

```

第三种算法是有关递归的，存在着大量的重复计算，图14展示了计算 F_4 递归调用的过程。我们可以对图14中的树进行遍历，进而得到内存上栈的操作顺序。在递归过程和栈操作中，例如 F_1 被计算了三次，存在非常多的重复计算。因此，第三种算法的复杂度很高： $T(n) = T(n-1) + T(n-2) + O(1)$ ，是指数级别的： $T(n) = O\left(\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]\right)$ 。

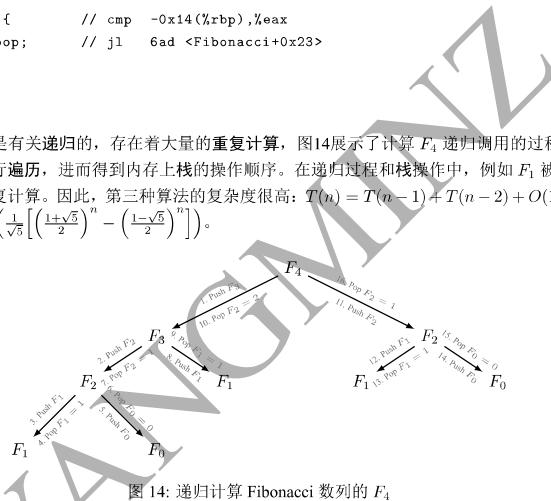


图14: 递归计算 Fibonacci 数列的 F_4

```

1 int Fibonacci(int n)
2 {
3     if (n == 0) return 0;
4     if (n == 1) return 1;
5     return Fibonacci(n - 1) + Fibonacci(n - 2);
6 }

```

对比第二种算法和第三种算法，其实我们已经隐约发现，尽管计算复杂度相差很大，但循环与递归之间存在某种可计算意义上的等价。我们不禁思考，怎样的递归与循环可以直接等价。

考虑上一节中复合函数 $h(g(\cdot))$ 的过程，我们发现，如果不要求被调函数 h 重进入到调用函数 h ，那么 g 的栈帧是可以不被保留的。这种递归的情况被称为尾递归 (Tail Recursion)。我们考虑 C 语言与机器指令的情况。递归过程 P_h 在它的尾部调用自身，跳转到 P_h 的起始地址 I_h ，也即最后一个指令是 `call h` 或 `jmp I_h`，那么这就是尾递归的情形。C 语言的编译器察觉到这一情况，通常会将尾递归优化成循环，进而减少栈帧传递参数的开销。而对于某些无法被简单优化为循环的递归调用，这时我们需要显式地在代码中创建栈，这种情况下循环的性能未必优于递归。甚至，设计不当的栈结构会频繁地在堆上申请 (`malloc`) 与释放 (`free`) 动态内存，性能反而可能差于栈上的递归调用。从状态机的角度看，循环或迭代 (Iteration)

的过程维护了一组有限的状态 S_i ，在迭代 i 中，维护与更新状态 $S_{i+1} \leftarrow S_i$ 。而在递归中，状态 S_i 则可以被视为递归深度 $n - i$ 时所传入的参数。

上述讨论从 λ 表达式的角度看会更加清晰，我们主要考虑 λ 表达式中的循环与递归。

首先考虑条件选择，在 C 语言中，条件选择的语法是这样的： $S \rightarrow \text{if}(Bool)\{S\}\text{else}\{S\}$ 。该语法根据计算得到的 $Bool$ 值去选择执行第一段代码 S_1 (`if` 内) 或第二段代码 S_2 (`else` 内)。因此，在 λ 表达式中，我们考虑 $Bool$ 的结果，也就是选择 S_1 或 S_2 。那么，真值 (`True`) 的结果为选择 S_1 : $\text{True} \equiv (\lambda S_1.(\lambda S_2.S_1))$ ，假值 (`False`) 的结果为选择 S_2 : $(\text{False} \equiv (\lambda S_1.(\lambda S_2.S_2)))$ 。在这里，我们再次运用了数据是函数的抽象的思想，将 $Bool$ 值考虑为函数，而非内存上的数据。

关于循环，我们可以在 λ 表达式中构建 List 的数据结构，进而实现。实际上，二十世纪五十年代的早期编程语言 Lisp (List Processing) 主要是用 List 的数据结构实现 λ 表达式。首先，我们考虑只有两个函数元素 e_0, e_1 的 List: $List_2 \equiv \lambda e_0.(\lambda e_1.(\lambda G.(G[e_0][e_1])))$ ，对 $List_2$ 取第一个函数的函数是: $Get_0 \equiv \lambda L.(L[\lambda e_0.(\lambda e_1.e_0)])$ ，取第二个函数: $Get_1 \equiv \lambda L.(L[\lambda e_0.(\lambda e_1.e_1)])$ 。我们举一个例子，给出更详细的解释。对于元素函数为 Church 数 N_0 与 N_2 的 List:

$$\begin{aligned}
 List_2[N_0, N_2] &\equiv (\lambda e_0.(\lambda e_1.(\lambda G.(G[e_0][e_1]))))[N_0][N_2] \\
 &\stackrel{e_0 \rightarrow N_0}{=} (\lambda e_1.(\lambda G.(G[N_0][e_1])))[N_2] \\
 &\stackrel{e_1 \rightarrow N_2}{=} \lambda G.(G[N_0][N_2])
 \end{aligned}$$

那么取 $List_2[N_0, N_2]$ 中的第二个函数:

$$\begin{aligned}
 Get_1[List_2[N_0, N_2]] &\equiv (\lambda L.(L[\lambda e_0.(\lambda e_1.e_1)]))[List_2[N_0, N_2]] \\
 &\stackrel{L \rightarrow List_2[N_0, N_2]}{=} (List_2[N_0, N_2])[\lambda e_0.(\lambda e_1.e_1)] \\
 &\equiv (\lambda G.(G[N_0][N_2]))[\lambda e_0.(\lambda e_1.e_1)] \\
 &\stackrel{G \rightarrow (\lambda e_0.(\lambda e_1.e_1))}{=} (\lambda e_0.(\lambda e_1.e_1))[N_0][N_2] \\
 &\stackrel{e_0 \rightarrow N_0}{=} (\lambda e_1.e_1)[N_2] \\
 &\stackrel{e_1 \rightarrow N_2}{=} N_2
 \end{aligned}$$

这样，我们用 λ 表达式就可以用函数 Get_1 对函数 $List_2$ 归约到它的第二个元素函数 N_2 。更长的 List 可以用 $List_2$ 以链表的方式实现: $List_2$ 的第一个函数 e_0 用来表示当前的值，第二个函数 e_1 用来指向下一个 $List_2$ 。在现实生活中，使用集成开发环境 (IDE) 的读者可以对链表/src/common/linkedlist.c 进行断点 (Breakpoint)，对 `next` 指针向下展开，这个过程和我们上述的 β 归约是非常相似的。到此为止，我们在一个长度为 n 的 List，对每一个 $List_2$ 节点进行 Get_1 操作，就能遍历整个 List，也就实现循环了。

最后，我们考虑递归函数。 λ 表达式在递归上有特殊的困难，因为每一个 λ 表达式是匿名的 (Anonymous)。在这之前，我们所取的一切名称，如 $N_0, N_1, List_2, Get_0, Get_1$ ，它们都不涉及自身，因此可以被替换、展开、归约。但是对于递归函数， λ 表达式无法替换它自身。为了解决这一困难，Turing 提出了不动点组合子 (Fixed-Point Combinator)，现在通常被称为 Y 组合子 (Y Combinator):

$$Y \equiv \lambda f.((\lambda x.f[x[x]])[\lambda x.(f[x[x]])])$$

它的推导需要 μ 算子对最小值进行搜索。考虑递归函数 $h \equiv \lambda x.E$, 其中表达式 E 包含了对 h 的引用。我们设外层函数 $g: g \equiv \lambda f.(\lambda x.E)$ (正如我们在部分递归函数生成时所考虑的), 用 μ 算子去搜索 h 对 g 的不动点 ($g[h] \rightarrow h$): $\mu_h^{g[h] \rightarrow h}$ 。对于这个搜索, 我们有必要加以特别的解释: 第一个解释是不动点约束条件 $g[h] \rightarrow h$, 它表明对于外层 λ 函数 $g \equiv (\lambda \text{Var}.E)$, 将变量 Var 替换为 h (包括 E 中出现的) 以后 (β 归约), 进行有限次变换, 能得到原函数 h 。也就是说, 我们搜索的是第一个能将 $g[h]$ 归约到 h 的函数 h 。

第二点解释是有关函数的序号。由于 μ 算子是对自然数 \mathbb{N}_0 的搜索, 而这里 $\mu_h^{g[h] \rightarrow h}$ 的搜索对象是函数 h , 这一点尤其让人困惑。实际上, 我们之所以能对函数进行搜索, 是因为存在一种映射将任意函数映射为一个自然数, 并且满足不同形式的函数 (符号) 会被映射到不同的自然数。这个映射是 Godel 根据质数性质所提出的, 对任意 URM 程序 P , 部分递归函数 f , λ 表达式 E , 都存在这样一个映射, 所得的自然数被称为 Godel 配数。有了 Godel 配数, 我们就可以对 λ 表达式进行排序和搜索, 寻找 h 了。

最后, 我们建立了有关 g 的匿名函数, 定义它为 \mathbf{Y} 组合子: $\mathbf{Y} \equiv \lambda g.\mu_h^{g[h] \rightarrow h}$ 。在 λ 表达式中, μ 算子有它的等价形式, 我们也不在这里讨论。总之, \mathbf{Y} 组合子最终呈现的形式, 就如我们先前所见到的。这样, 我们利用函数的不动点, 也就在 λ 演算中实现了递归。 \mathbf{Y} 组合子是不设置递归终止的, 可以终止的被称为 \mathbf{Z} 组合子。

Cutland 的 [[cutland1980computability](#)] 主要从计算机的视角去介绍计算理论, 我们在本章和 Cutland 采用了同样的观点, 以 URM——而非 Turing 在二十世纪三十年代提出的计算机 (Turing Machine, TM)——为核心, 去讨论状态机的计算能力, 这是由于 URM 结构简洁。关于 Cutland 这本著作, 在世纪初的中文互联网上还曾有过这样一段轶事。现南京大学的喻良教授在南大博士毕业之后写了一篇博客, 回顾五年的研究生活, 行文非常诙谐, 充满了独特的魅力: “[1998年的年末, 我来到了南京。\(后略\)](#)” 喻良教授刚开始读研究生时, 看的就是 Cutland 的递归论。立志做学术研究的读者可以读一读这篇博客, 了解做学术的品味, 想必会受益很多。

关于执行 Von Neumann 体系结构计算机指令所需的硬件资源, [[bryant2003computer](#)] 介绍了许多。读者可以阅读体系结构方面的经典著作——Hennessy 与 Patterson 的 [[hennessy2011computer](#)], 两位作者共同获得了 2017 年度的 ACM 图灵奖——从而对体系结构有更深刻的理解。我们反复强调, 在 Von Neumann 体系结构中, “指令即数据” 是一个非常重要的思想, 也是过程调用与递归函数得以实现的基础。在 Church 的 λ 演算中, 反过来, “数据即函数”, 也是同等重要的。关于 λ 演算, Stuart 的 [[stuart2013understanding](#)] 是用 Ruby 来介绍计算理论的, 展露了许多 λ 表达式的技巧。MIT 的经典教科书 SICP [[abelson1996structure](#)] 则用 Lisp 的方言 Scheme 来阐述函数式编程, 充分利用函数进行抽象。如果读者平时大多使用命令式的编程语言, 那么刚开始接触函数式的思想时, 一定感到自己进入了一个全新的世界。如果读者追寻更数学的理解, 应当阅读 Kleene 有关 Metamathematics 的著作 [[kleene1952introduction](#)], 系统地了解 λ 演算与 Godel 不完备定理的证明。

部分递归函数是数学语言的, λ 演算是符号系统的, URM 与 TM 则暗含了计算机的物理实现, 之后发展出 Von Neumann 体系结构, 并逐步建立了我们今天所见的计算机系统。这一章我们主要考虑的还是实现字符串的指令集, 只是为了粗略了解指令集设计者的思想, 我们才回顾历史, 讨论计算模型之间的关联。关于字符串的指令集, 我们使用了一些语法分析的技巧, 这些内容来自于编译原理, 可以参考 Alfred 的 [[aho1986compilers](#)], Alfred 也于近期获得了 2020 年度的图灵奖。这本书是编译器领域的经典著作, 因为封面是一只火龙, 因此也被称为龙书 (Dragon Book)。最后, 我们关于指令集的近似实现, 最权威的参考资料还是 Intel 公司编纂的 CPU 编程手册 [[guide2011intel](#)]。