1 数据类型

计算机系统的计算能力，主要来自于**储存**与**运算**。其中储存能力是由物理内存提供的，运算能力是由中央处理器提供的。计算机对于数据的运算，主要就是对于储存在内存上的数据进行计算。而数据本身，则是对于一系列 {"0", "1"} 排列的编码。

现在，我们开始考虑对 {"0", "1"} 的编码。首先认为，现实中的物理内存可以被看做一系列**有限长度**（例如 $N \in \mathbb{N}_0$）的 {"0", "1"} 序列，也即：

$$m_0, \quad m_1, \quad \cdots, \quad m_i, \quad m_{i+1}, \quad \cdots m_{N-1}, \quad m_i \in \{\texttt{"0"}, \texttt{"1"}\}$$

同理，我们可以对任意有限的字符集合 $\Sigma_{|\Sigma|<\infty}$ 定义任意长度的序列：

$$s_0, \quad s_1, \quad \cdots, \quad s_i, \quad s_{i+1}, \quad \cdots s_{N-1}, \quad s_i \in \Sigma$$

对于这样有限长度的序列，其所能表达的**不同的**序列总数为 $|\Sigma|^N$。事实上，对于任意有限的字符集 Σ，其**可数**（countable）长度的元素序列（也就是单词），也构成了可数大小的集合：

$$\{W_0 = \varnothing, \quad W_1 = \Sigma, \quad W_2 = \Sigma \times \Sigma, \quad \cdots \quad W_n = \Sigma^n, \quad \cdots\}$$

其中二元运算 $L \times \Sigma$ 是对所有左集合 L 的元素的尾部添加右侧字符集 Σ 中的所有元素。例如 $L = \{aa, ab, bb\}$，$\Sigma = \{a, b\}$，则 $L \times \Sigma = \{aaa, aba, bba, aab, abb, bbb\}$。

注意，所有可数长度序列的集合，按照序列长度的排序（$N = 0, 1, 2, \cdots$），其本身也是可数的。我们这样定义序列的序号：对于有限字符集 Σ 本身，以任意一种方法排序，列出它的元素：$\Sigma = \{s_0 \prec s_1 \prec \cdots \prec s_{|\Sigma|-1}\}$。假定空集 $\varnothing$ 中的元素（即无元素）标号为 0，单个元素集合 Σ 的标号，按照 $\{s_0 \prec s_1 \prec \cdots \prec s_{|\Sigma|-1}\}$ 排序，分别为 $\{1, 2, \cdots, |\Sigma|\}$。

接下来，我们用数学归纳的方法给定任意长度的单词集合的排序号。对于长度为 $N \in \mathbb{N}$ 的单词，其所有元素的集合为 W_N，其中单词元素的序号为：

$$t, t+1, \cdots, t + |\Sigma|^N - 1$$

则 $W_{N+1} = W_N \times \Sigma$ 可以按照 $\{s_0 \prec s_1 \prec \cdots \prec s_{|\Sigma|-1}\}$ 的顺序，对 W_N 中的每个元素标号，每次标号后将序数加一。例如对于字符集 $\Sigma = \{a \prec b \prec c\}$，其所生成的序列序号如下：

$$\varnothing \to 0, a \to 1, b \to 2, c \to 3, aa \to 4, ab \to 5, ac \to 6, ba \to 7, \cdots$$

到此为止，我们已经建立了一个一对一的映射 f，可以将任意一个**排序的**有限字符集 Σ 所生成的任意序列 s 映射到唯一的一个序号中：

$$f : (\varnothing \cup \Sigma \cup \Sigma^2 \cup \cdots) \ni s \mapsto i \in \mathbb{N}_0$$

这样的映射为我们解释序列创造可能。我们只需要做两步映射即可，对于任一有限长度的序列 s，首先用 f 映射出其序号 $f(s) \in \mathbb{N}_0$，其次对这个序号 $f(s)$ 做出解释 g，这样我们就创造了一种具有含义与解释的语言。需要特别注意的是，我们只是对字符集到单词建立了可数的映射，实际上语义与逻辑本身也可以用序数表达，请参见 Kurt Godel 对**不完备定理**（Incomplete Theorem）的证明。

下面我们来观察一个具体的例子。对于有限字符集 $\Sigma = \{\oplus \prec \odot \prec \otimes\}$，我们有序数映射 $f(\cdot)$。对于自然数序数，我们定义这样一个“解释表”或“含义表”：

$$\cdots, g(5) = \text{eat}, \cdots, g(9) = \text{I}, g(10) = \text{apple}, \cdots$$

这样，我们便已经建立了 $\Sigma = \{\oplus \prec \odot \prec \otimes\}$ 到一种自然语言（英语）的映射，也即对每一种序列做出了解释。在 $\{\oplus \prec \odot \prec \otimes\}$ 语言下，序列 $\oplus\odot$ 被解释为 eat，$\odot\otimes$ 被解释为 I，$\otimes\oplus$ 被解释为 apple，而三个序列的组合也就此有了可解释的含义，I eat apple：$\odot\otimes \quad \oplus\odot \quad \otimes\oplus$。

这个例子为我们提供了一个观点，即**"信息来自于表达与差异"**。对于任意字符集，序列的差异使我们能够用序号映射 $f(\cdot)$ 去标注它们之间的不同，而对于差异的解释 $g(\cdot)$ 则创造了序列的**语义**。实际上，假设长度 n 的定长（Fixed-Length）序列中，不同的序列是等可能地出现的，那么，一共有 $|\Sigma|^n$ 种可能，每一个序列出现的概率也就是 $p = \frac{1}{|\Sigma|^n}$。根据 Claude Shannon，那么这个定长序列的表达能力可以通过**熵**（Entropy）来描述，其中熵是每个序列统计意义上的平均信息量：

$$H = \sum_{i=1}^{|\Sigma|^n} p \log_2\left(\frac{1}{p}\right) = |\Sigma|^n \cdot \frac{1}{|\Sigma|^n} \cdot \log_2(|\Sigma|^n) = n \log_2(|\Sigma|)$$

如果我们拥有某种语言的词典，从第一页的第一个单词开始，遍历词典中所有不重复的单词，包括这些单词的任意屈折变态，以这个遍历作为单词表 $g(\cdot)$，我们便可以用任意一个符号系统 Σ 表达该语言。

一个极端的假设是这样的，某个字符集为单个元素：$\Sigma = \{\sigma\}$，那么我们序号映射 $f(\cdot)$ 的结果也就是序列的长度本身。如果用 $\{\sigma\}$ 系统去编写词典，那一定很耗费纸张。如果我们制作 $\{\sigma\}$ 类型的计算机，那么程序的运行则需要等待相当长的时间，或需要相当大的空间，因为 $\{\sigma\}$ 计算机会通过"（时间的或空间的）长度"来区分信息。从**熵**的角度看，每一个定长序列的信息量都是零，也即确定的，不含任何信息：$n \cdot \log_2(1) = 0$。

幸而我们的计算机是 {"0", "1"} 类型的，所以一个固定长度（N）的序列可以表达 2^N 种类的信息（序列）。这样，定长编码的**熵**就是：$n \log_2(2) = n$。在我们理解计算机系统或 C 语言时，便会发现计算机系统正是通过这样的方式来组织 {"0", "1"} 序列的。不同长度的 {"0", "1"} 序列序列可以用不同的方法去解释，这也就构成了**基本数据类型**。

1.1 整数类型

从上述讨论中，我们已经建立了从有限字符集 Σ 生成可数序列的过程。对计算机的集合 $\Sigma = \{\texttt{"0"} \prec \texttt{"1"}\}$ 应用这个过程：

$$\begin{aligned}
&W_0 = \varnothing \\
&W_1 = \{\texttt{"0"} \prec \texttt{"1"}\} \\
&W_2 = \{\texttt{"00"} \prec \texttt{"01"} \prec \texttt{"10"} \prec \texttt{"11"}\} \\
&W_3 = \{\texttt{"000"} \prec \texttt{"001"} \prec \texttt{"010"} \prec \texttt{"011"} \prec \texttt{"100"} \prec \texttt{"101"} \prec \texttt{"110"} \prec \texttt{"111"}\} \\
&\cdots
\end{aligned}$$

可以看到，对于每一个定长的序列集合 W_N，其序列元素按照序号 $f(s)$ 排序，计算其对偏序最小序列 $s_0 = \texttt{"00}\cdots\texttt{0"}$ 的偏置：

$$g : \mathbb{N}_0 \ni f(s) \mapsto f(s) - f(s_0) \in \mathbb{N}_0$$

我们观察到这个解释映射（即一种对序数 $f(\cdot)$ 定义的 $g(\cdot)$）与二进制下的整数数值对加法是**同构**（Isomorphism）的：

$$BU : W_N \ni s \mapsto \sum_{i=0}^{N-1} s[i] \times 2^i, \quad s[i] \in \{0, 1\}$$

这样，我们对序数建立了新的解释，任何一个定长的"0"，"1"序列可以被解释为一个二进制的自然数：

$$\begin{aligned}&\texttt{"0"} \to 0(0), \quad \texttt{"1"} \to 1(1)\\&\texttt{"00"} \to 00(0), \quad \texttt{"01"} \to 01(1), \quad \texttt{"10"} \to 10(2), \quad \texttt{"11"} \to 11(3)\\&\ldots\end{aligned}$$

我们所建立的这个映射，即是**无符号整数**（Unsigned Integer）。需要特别注意的是，该映射不仅仅定义了序列到自然数之间的映射，同时通过序号 $f(\cdot)$ 定义了序列的加法，对二进制的数值保持同构。而一个定长的"0"，"1"序列，也就是一个数据类型。

> 在计算机系统或 C 语言中，上述讨论的每一个字符"0"及"1"被称为**二进制位**（Bit），而每 8 个 Bit 构成一个字节（Byte），也即两个十六进制位。通常，每 32 或 64 个 Bit 构成计算机对无符号整数的计算。在 C 语言中，标准库 `stdint.h` 提供了直接指定 Bit 数量的数据类型：`uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`。

对于**有符号整数**（Signed Integer），即包括负数，我们需要对序数 $f(\cdot)$ 建立新的解释 $h(\cdot)$。只要该解释映射 $h(\cdot)$ 对 $f(\cdot)$ 在加法上也同构，那么有符号整数与无符号整数两种解释也是同构的。

为了均匀分配非负整数与负整数的数量，我们按照最高位来对 2^N 个序列进行划分：最高位 $s[N-1]$ 为 1 的取负数，最高位为 0 的取非负整数。具体的映射是这样的，这也被称为**2 的补码**（Two's Complement）：

$$BS : W_N \ni s \mapsto (-1) \times s[N-1] \times 2^{N-1} + \sum_{i=0}^{N-2} s[i] \times 2^i, s[i] \in \{0, 1\}$$

不难发现，这个映射也对序号 $f(\cdot)$ 与加法是同构的。以 $N = 3$ 为例，我们考察 3-Bit 的有符号整数与无符号整数，如图1。序列 s 按照 $f(\cdot)$ 连续排布，可以看到，在加法和减法运算下，无符号数与有符号数分别有一处是不连续的，这些部分被称为**溢出**（Overflow）。

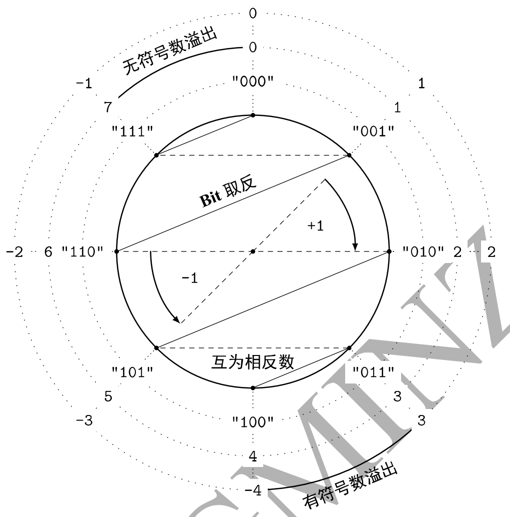

图 1: 3 位有符号数与无符号数

事实上，无符号整数的溢出与 CPU 中加法器的硬件实现相关。我们仍然以 3-Bit 的无符号整数为例：

$$
\begin{array}{rcccc}
 & & 1 & 1 & 1 \\
+ & & 1 & 1 & 0 \\
\hline
 & 1 & 1 & 0 & 1
\end{array}
$$

可以看到，两个 3-Bit 无符号整数的加法产生了一个 4-Bit 无符号整数。在 CPU 内部的加法器中，每一个 Bit 占据了一个物理位，因此最高位产生的进位将被直接舍弃。这样，物理上也就发生了无符号整数的溢出。对二进制数舍弃最高位，使得我们的无符号整数本质是在对加法运算的结果**取模**（Modulus），对乘法也如此。这样，我们其实已经建立了一个模 2^N 的**代数结构**（Algebra）。

> 在 64 位的计算机系统中，我们通常可以用 `uint64_t` 来描述 Bit 序列的结构。对于 32-Bit 和 64-Bit 无符号与有符号整数的情况，都是上述情况的推广。

对于整数在内存上的储存，我们需要特别加以解释。物理内存总是以 8-Bit，也即 Byte 为单位的。但在不同的机器中，访问 Byte 的顺序并不相同。通常我们能接触到的机器被称为**小端机**（Little Endian），小端机总是将低位的 Byte 存放在内存的低地址，高位的 Byte 存放在高地址。例如 `0x12345678`，其中 Byte 从低位到高位分别为:`0x78`，`0x56`，`0x34`，`0x12`,这也就是它们在内存上从低地址到高地址被储存的顺序。

最后，我们给出对无符号、有符号数加减法运算是否溢出的判断。考虑加法（*Type*）`v` $\leftarrow$（*Type*）`a +`（*Type*）`b` 以及减法（*Type*）`v` $\leftarrow$（*Type*）`a -`（*Type*）`b`。首先考虑（*Type*）为 `uint64_t` 的情形，也即无符号数的加法与减法。此时，判断加法溢出为 `(v < a)`；判断减法溢出为 `(v > a)`。

再考虑（*Type*）为 `int64_t` 的情形，也即有符号数的加法与减法。对于有符号数，我们需要考虑最高位，也即符号位的情况。加法溢出的情况是这样的：`((sa == 0) && (sb == 0) && (sv == 1)) || ((sa == 1) && (sb == 1) && (sv == 0))`，这可以被化简为***`(!(sa ^ sb) && (sb ^ sv))`***。减法溢出的情况是这样的：`((sa == 0) && (sb == 1) && (sv == 1)) || ((sa == 1) && (sb == 0) && (sv == 0))`，这可以被化简为***`((sa ^ sb) && !(sb ^ sv))`***。

1.2 浮点数类型

二进制下浮点数的表示问题，其实就是如何表达一个有限位带小数的数字。我们将二进制的 `{'0', '1'}` 序列看作一个**矢量**（Vector），最左侧为低位 Bit，最右侧为高位 Bit，按照下标增长：

$$\mathbf{a} = [0, 1, \cdots, 0, 1] \in \{0, 1\}^N$$

那么，关于无符号整数的数值，其实就是该序列矢量与一个**基向量**（Base）的**数量积**（Scalar Product）：

$$\mathbf{a} \cdot \mathbf{b} = [0, 1, \cdots, 0, 1] \cdot [2^0, 2^1, \cdots, 2^{N-1}]^T = \sum_{i=0}^{N-1} \mathbf{a}[i] \times 2^i$$

关于一个有限小数的表达，我们只需要“收缩”基向量即可：

$$\mathbf{b}' = 2^{-K} \cdot [2^0, 2^1, \cdots, 2^{N-1}] = [2^{-K}, 2^{1-K}, \cdots, 2^{N-1-K}]$$

$$\mathbf{a} \cdot \mathbf{b}' = [0, 1, \cdots, 0, 1] \cdot [2^{-K}, 2^{1-K}, \cdots, 2^{N-1-K}]^T = \sum_{i=0}^{N-1} \mathbf{a}[i] \times 2^{i-K}$$

这样，我们就可以将带小数点的二进制串解释为一个有限小数了：对于二进制串的部分，也就是基向量各个部分 $\mathbf{b}'[i]$ 的系数 $\mathbf{a}[i]$，我们予以保留；根据小数点对最低位 [0] 的不同偏移 K，我们采用不同收缩程度 2^{-K} 的基向量 $\mathbf{b}'$：

$$\mathbf{a} \cdot \mathbf{b}' = 1 \times 2^{N-1-K} \quad \cdots \quad 1 \times 2^1 \quad 0 \times 2^0 \quad \bullet \quad 1 \times 2^{-1} \quad 0 \times 2^{-2} \cdots \quad 1 \times 2^{-K}$$

接下来，我们对 3-Bit 的二进制序列来枚举所有可能的浮点数，来给读者最直观的印象：

0×2^2	0×2^1	$0 \times 2^0 \bullet$	0×2^1	$0 \times 2^0 \bullet 0 \times 2^{-1}$	$0 \times 2^0 \bullet 0 \times 2^{-1}$	0×2^{-2}	$\bullet 0 \times 2^{-1}$	0×2^{-2}	0×2^{-3}
0×2^2	0×2^1	$1 \times 2^0 \bullet$	0×2^1	$0 \times 2^0 \bullet 1 \times 2^{-1}$	$0 \times 2^0 \bullet 0 \times 2^{-1}$	1×2^{-2}	$\bullet 0 \times 2^{-1}$	0×2^{-2}	1×2^{-3}
0×2^2	1×2^1	$0 \times 2^0 \bullet$	0×2^1	$1 \times 2^0 \bullet 0 \times 2^{-1}$	$0 \times 2^0 \bullet 1 \times 2^{-1}$	0×2^{-2}	$\bullet 0 \times 2^{-1}$	1×2^{-2}	0×2^{-3}
0×2^2	1×2^1	$1 \times 2^0 \bullet$	0×2^1	$1 \times 2^0 \bullet 1 \times 2^{-1}$	$0 \times 2^0 \bullet 1 \times 2^{-1}$	1×2^{-2}	$\bullet 0 \times 2^{-1}$	1×2^{-2}	1×2^{-3}
1×2^2	0×2^1	$0 \times 2^0 \bullet$	1×2^1	$0 \times 2^0 \bullet 0 \times 2^{-1}$	$1 \times 2^0 \bullet 0 \times 2^{-1}$	0×2^{-2}	$\bullet 1 \times 2^{-1}$	0×2^{-2}	0×2^{-3}
1×2^2	0×2^1	$1 \times 2^0 \bullet$	1×2^1	$0 \times 2^0 \bullet 1 \times 2^{-1}$	$1 \times 2^0 \bullet 0 \times 2^{-1}$	1×2^{-2}	$\bullet 1 \times 2^{-1}$	0×2^{-2}	1×2^{-3}
1×2^2	1×2^1	$0 \times 2^0 \bullet$	1×2^1	$1 \times 2^0 \bullet 0 \times 2^{-1}$	$1 \times 2^0 \bullet 1 \times 2^{-1}$	0×2^{-2}	$\bullet 1 \times 2^{-1}$	1×2^{-2}	0×2^{-3}
1×2^2	1×2^1	$1 \times 2^0 \bullet$	1×2^1	$1 \times 2^0 \bullet 1 \times 2^{-1}$	$1 \times 2^0 \bullet 1 \times 2^{-1}$	1×2^{-2}	$\bullet 1 \times 2^{-1}$	1×2^{-2}	1×2^{-3}

以上就是二进制小数的全部基础，正如“浮点（Float Point）”这一名字所暗示的，小数点的位置是浮动的，例如 000. = 00.0 = 0.00 = .000，这样的表达是冗余的，因为小数点的位置没有被**对齐**，因此我们需

要对它进行处理。例如 `float` 类型，我们要在 32 位中表达小数，就是对上述的二进制小数采用**科学计数法**（Scientific Notation）。就像我们在小学或初中课本中所学到的，科学计数法所做的，其实是对齐小数点的位置。在采用科学计数法以后，忽略前导零，二进制小数可以通过**符号**（Sign）、**指数**（Exponent）、**尾数**（Fraction）来表达。并且，为了保证既可以表示极大的小数（正指数），同时也可以表达极小的小数（负指数），我们要对指数减去一个常量作为偏置（Bias）：

$$(-1)^{\mathbf{S}} \times (00\cdots01.\mathbf{F}) \times 2^{\mathbf{E}-Bias}$$

在这里，2^0 位上的 1 是可以缺省（Default）的，这样可以节省浮点数中的 Bit 位。以 `float` 为例，32 位的 `{0, 1}` 序列 a 被划分为下列区域：

符号（Sign）	指数（Exponent）	尾数（Fraction）
a_{31}	$a_{30}, \cdots, a_{23}$	$a_{22}, \cdots, a_0$
1	8	23

在此基础上，我们可以将浮点数分为四类：

第一类：规格化的（Normalized）

符号	指数	尾数
a_{31}	$0 < a_{30}\cdots a_{23} < 2^8$	$a_{22}a_{21}\cdots a_0$

规格化的浮点数的指数部分既非全零（`0,0000,000`），亦非全一（`1,1111,111`）。它采用我们先前所描述的方法去表达浮点数。需要注意的是，由于指数位有 8 Bits，因此我们用 $2^{8-1}-1=127$ 作为偏置，平衡指数取正负数的数量：2^{E-127}。这样，浮点数的各个部分可以如下计算：

a_{31}	$(-1)^{a_{31}}$	符号位
$a_{30}\cdots a_{23}$	$\sum_{i=23}^{30} a_i \times 2^{i-23} - 127$	指数
$a_{22}a_{21}\cdots a_0$	$1+\sum_{i=0}^{22} a_i \times 2^{i-23}$	缺省为 1 的尾数 $1.a_{22}a_{21}\cdots a_0$

$$Value = (-1)^{a_{31}} \times \big(1+\sum_{i=0}^{22} a_i \times 2^{i-23}\big) \times 2^{\sum_{i=23}^{30} a_i \times 2^{i-23} - 127}$$

第二类：非规格化的（Denormalized）

非规格化的浮点数的指数部分为 0，因此它的指数是固定的，也即 -127。因此，非规格化的浮点数用来表示非常小的数值，也就不需要将 2^0 置为 1 了。这样，非规格化的浮点数可以如下表示：

$$(-1)^{\mathbf{S}} \times (00\cdots00.\mathbf{F}) \times 2^{-Bias}$$

a_{31}	$(-1)^{a_{31}}$	符号位
`0,0000,000`	-127	指数
$a_{22}a_{21}\cdots a_0$	$\sum_{i=0}^{22} a_i \times 2^{i-23}$	缺省为 0 的尾数 $0.a_{22}a_{21}\cdots a_0$

$$Value = (-1)^{a_{31}} \times \big(\sum_{i=0}^{22} a_i \times 2^{i-23}\big) \times 2^{-127}$$

第三类：无穷大（Infinity）

对于无穷大数，我们用符号位指示其为正无穷大（$+\infty$）或是负无穷大（$-\infty$）。它的指数部分为全一，而尾数部分为全零：

符号	指数	尾数
a_{31}	111,1111,1	000,0000,0000,0000,0000,0000

第四类：非数（NaN）

除此以外的，也即指数部分取到全一，而尾数部分非零的，在浮点数格式中被认为非数（Not a Number）：

符号	指数	尾数
a_{31}	111,1111,1	$\neq$ 000,0000,0000,0000,0000,0000

以上我们所描述的 32 位浮点数格式，可以用下面的数据结构来表示：

/src/headers/cpu.h

```
typedef union
{
    uint32_t        __float_bits;
    struct
    {
        uint32_t    frac    : 23;
        uint32_t    expo    : 8;
        uint32_t    sign    : 1;
    };
} float32_t;
```

对于科学计数法，当尾数部分不足够时（这通常出现在类型转换中，例如 (float)0xffffabcd)，我们需要对原数进行**近似**（Rounding），这也就涉及到浮点数的**精度**（Precision）问题。例如，我们要将某大数值的 uint32_t 类型的无符号整数转换为同样**数值**的浮点数 float：

$$1a_{30}\cdots a_1a_0 = 1a_{30}\cdots a_1a_0.0\times 2^0 = 1.a_{30}\cdots a_1a_0\times 2^{31}$$

显然，当前我们有 31 位的尾数，而这超过了规格化的浮点数所能表达的精度，也就是 23 位尾数。因此，我们要舍弃最低的 8 位，才能用 float 去表示：

$$1.a_{30}\cdots a_8a_7a_6a_5a_4a_3a_2a_1a_0\times 2^{31}\approx 1.a_{30}\cdots a_8\xcancel{a_7a_6a_5a_4a_3a_2a_1a_0}\times 2^{31}$$

在浮点数中，近似的规则是这样的。我们先取三个 Bit，$\cdots.\cdots G|RS$，它们的含义如下

- G（Guard Bit），被保留的最低位，上例中的 a_8；
- R（Round Bit），被舍弃的最高位，上例中的 a_7；
- S（Sticky bit），在 R 位右侧所有被舍弃的低位之**或**（OR），上例中的 $\bigvee_{i=0}^{6} a_i$；

计算得到 G, R, S 以后，按照以下规则进行向下近似（$+0$）或向上近似（$+1$）：

G	R	S	近似	G	R	S	近似
0	0	0	$+0$	1	0	0	$+0$
0	0	1	$+0$	1	0	1	$+0$
0	1	0	$+0$	1	1	0	$+1$
0	1	1	$+1$	1	1	1	$+1$

实际上，这个规则可以被卡诺图（Karnaugh Map）简化：

	$RS = 00$	$RS = 01$	$RS = 10$	$RS = 11$
$G = 0$	$+0$	$+0$	$+0$	$+1$
$G = 1$	$+0$	$+0$	$+1$	$+1$

最后我们得到向上近似所需增加的数值（$+0$ 或 $+1$）的布尔表达式：

$$+(R \wedge (G \vee S)), \quad G, R, S \in \{0, 1\}$$

我们在`/src/common/convert.c`中实现了 `uint32_t` 到 `float` 的类型转换，其中有关浮点数近似的代码如下：

/src/common/convert.c

```
                                           // uint32_t u;
uint32_t G = (u >> (n - 23)) & 0x1;       // Guard    bit
uint32_t R = (u >> (n - 24)) & 0x1;       // Round    bit
uint32_t S = 0x0;                         // Sticky   bit
for (int j = 0; j < n - 24; ++ j)
{
    S = S | ((u >> j) & 0x1);
}
if ((R & (G | S)) == 0x1)
{
    // Rounding to G + 1
}
```

1.3 字符类型

字符类型是对 Byte 的另一种翻译。每一个 Byte，从 `00000000` 到 `11111111`，共有 256 种取值。在 C 语言中，对一个 Byte 取数据类型为**字符**（Character），那么这个 Byte 的数值会通过 ASCII 码表被解释为一个字符，例如 `0x30` 为`'0'`，`0x39` 为`'9'`，`0x41` 为`'A'` 以及 `0x5a` 为`'Z'`。

需要特别注意的是 `char` 在内存上的储存。和 `uint64_t` 以及 `float` 不同，每一个 `char` 是一个单独的 Byte，因此它的排布并不区分大端与小端。相比于单个字符，更常见的是 `char` 数组，一个 `char` 数组也就构成了一个**字符串**（String）：

```
char str[16] = {'\0'};  // will take 16 * 1 = 0x10 Bytes
```

字符数组在内存中按照下标从小到大，从低地址向高地址储存字符。对于字符串，内存上的每一 Byte 被解释为 `char`，直到遇到第一个字符串的终止符`'\0'`（`0x00`）。这说明，字符串所占用的内存空间与字符串本身的长度并不相同。在 C 语言中，库函数通过 `string.h` 提供了对字符串的操作。`size_t strlen(const char * str)` 会返回字符串的长度，计算到第一个`'\0'` 为止。而运算符 `sizeof` 则给出内存的占用。我们以 `char str[16]` 上的字符串`"mov %rsp,%rbp"` 为例，如图2。

strlen(str) = 13

'm'	'o'	'v'	' '	'%'	'r'	's'	'p'	','	'%'	'r'	'b'	'p'	'\0'	'\0'	'\0'
6d	6f	76	20	25	72	73	70	2c	25	72	62	70	00	00	00

sizeof(str) = 16

图 2: 字符数组

1.4 指针类型

到此为止，我们已经在内存上定义了各种数据类型。这些数据类型都在物理内存上对应唯一的位置（起始地址），被称为**物理地址**（Physical Address）。实际上，我们在运行程序时，通常真正使用的是**虚拟地址**，不过这个话题我们之后再讨论。总之，每一个数据类型在物理内存或（逻辑上的）虚拟内存上占据了空间，它的起始地址也就是其地址。如果我们将（物理或逻辑）内存看作一个巨大的 Byte 数组，那么这个地址也就是内存数组的下标（Index），它所存的数值也就是 Byte 值，从 `0x00` 到 `0xff`。

在这样的内存模型下，我们常常需要取得一个变量的（逻辑）地址，这也就是 C 语言中的**指针**（Pointer）类型。但在实践中，用“指针”这个词常常会使程序员感到困扰，分辨不清它的具体含义。为了简单起见，我们可以利用类型转换，将指针类型（`void *`）直接转换为一个（`uint64_t`）的无符号整数，也就是 Byte 数组的下标，这样会更加清晰易懂。实际上，这也是一些历史更悠久的编程语言所采用的方案：

```
uint8_t  value = 0xab;
uint64_t addr  = (uint64_t)&value;      // "&value" is taking address (pointer)
assert(*((uint8_t *)addr) == value);    // "*pointer" is taking value stored
                                        //      on virtual memory: mem[pointer]
```

利用指针，我们可以在 C 语言中实现近似**面向对象**（Object-Oriented）的编程模型的诸多特质。譬如对于**泛型**（Genericity），我们以实现一个**双向环形链表**为例，这个双向环形链表需要支持多种数据类型，包括并不限于 `int`，`char`，甚至在下文中提到的 `struct`。利用指针的特性，我们可以这样定义链表的节点：

/src/common/linkedlist.c

```
// circular doubly linked list
// this linked list can be implemented as a FIFO queue
typedef struct LINKED_LIST_NODE_STRUCT
{
    uint64_t    addr;
    struct      LINKED_LIST_NODE_STRUCT *prev;
    struct      LINKED_LIST_NODE_STRUCT *next;
} listnode_t;

typedef struct LINKED_LIST_STRUCT
{
    listnode_t  *head;
    uint64_t    counter;
} linkedlist_t;
```

链表节点所储存的值不再是变量的数值本身，而是该变量的**地址**。对于该变量地址的类型翻译，这份责任被委托给链表创建者本身。通过这样的定义，我们甚至可以实现对链表的**封装**（Encapsulation）：

/src/headers/common.h

```
// linked list
uint64_t    linkedlist_allocate     ();
void        linkedlist_free         (uint64_t listAddr_addr);
void        linkedlist_add          (uint64_t listAddr_addr, uint64_t value_addr);
void        linkedlist_delete       (uint64_t listAddr_addr, uint64_t value_addr);
uint64_t    linkedlist_count        (uint64_t listAddr_addr);
```

```
uint64_t    linkedlist_next         (uint64_t listAddr_addr);
```

链表的类型被定义在 linkedlist.c 之内，其他的.c 文件无法访问 listnode_t 或 linkedlist_t，这样就对数据结构本身实现了一次封装。链表对外只暴露了上述几个函数，数据结构并不关心链表本身和节点中的数据是被分配在内存的**栈**，**堆**或**数据段**上，或是它们的数据类型，只需要它们的地址（指针），链表都可以管理。例如，我们管理一个 int 类型的链表：

```
int a = 1, b = 2;                           // values are stored on stack
uint64_t listaddr = linkedlist_allocate();  // linked list is allocated on heap
                                            // but its heap address is stored on
                                            // stack to be referenced
linkedlist_add  (   (uint64_t)(&listaddr),  (uint64_t)(&a)  );
linkedlist_add  (   (uint64_t)(&listaddr),  (uint64_t)(&b)  );
linkedlist_free (   (uint64_t)(&listaddr)                   );
```

除此以外，在**链接**中我们还将看到，其实函数本身也具有地址，函数的地址就是其**汇编指令**的起始地址，参数传递在**编译**中都已经被处理为对**寄存器**和**内存**的读取，因此，我们其实甚至可以定义函数的指针，这样就可以将函数本身（其指针或地址）作为参数传递给其他函数，也就实现对函数的**回调**（Callback）。实际上，在下一章中，我们实现**指令集**时所采用的正是这种回调的方法。

> 在不同的体系结构中，一个指针（(void *)）所占据的内存大小也不相同。通常，在 32-Bit 的机器上，一个指针占据 32 位。类似的，指针在 64-Bit 的机器上占据 64 位。尽管如此，指针的高位 Bit 通常都是零，这是因为虚拟地址本身并不足占据全部 64 位。让我们来观察一些具体的例子：

64 位整数数组以及内存布局：

```
int64_t a[2][2] = { { 0x0123456789abcdef, 0xffffffffffffffff },
                    { 0x7fffffffffffffff, 0x8000000000000000 } };
```

0x7ffffffee160 : 0xef	0x7ffffffee168 : 0xff	0x7ffffffee170 : 0xff	0x7ffffffee178 : 0x00
0x7ffffffee161 : 0xcd	0x7ffffffee169 : 0xff	0x7ffffffee171 : 0xff	0x7ffffffee179 : 0x00
0x7ffffffee162 : 0xab	0x7ffffffee16a : 0xff	0x7ffffffee172 : 0xff	0x7ffffffee17a : 0x00
0x7ffffffee163 : 0x89	0x7ffffffee16b : 0xff	0x7ffffffee173 : 0xff	0x7ffffffee17b : 0x00
0x7ffffffee164 : 0x67	0x7ffffffee16c : 0xff	0x7ffffffee174 : 0xff	0x7ffffffee17c : 0x00
0x7ffffffee165 : 0x45	0x7ffffffee16d : 0xff	0x7ffffffee175 : 0xff	0x7ffffffee17d : 0x00
0x7ffffffee166 : 0x23	0x7ffffffee16e : 0xff	0x7ffffffee176 : 0xff	0x7ffffffee17e : 0x00
0x7ffffffee167 : 0x01	0x7ffffffee16f : 0xff	0x7ffffffee177 : 0x7f	0x7ffffffee17f : 0x80
a[0][0] = 1	a[0][1] = -1	a[1][0] = MaxVal	a[1][1] = MinVal

对它每一个元素的指针引用：

```
int64_t *p[4] = { &a[0][0], &a[0][1], &a[1][0], &a[1][1] };
```

0x7fffffffee180 : 0x60	0x7fffffffee188 : 0x68	0x7fffffffee190 : 0x70	0x7fffffffee198 : 0x78
0x7fffffffee181 : 0xe1	0x7fffffffee189 : 0xe1	0x7fffffffee191 : 0xe1	0x7fffffffee199 : 0xe1
0x7fffffffee182 : 0xfe	0x7fffffffee18a : 0xfe	0x7fffffffee192 : 0xfe	0x7fffffffee19a : 0xfe
0x7fffffffee183 : 0xff	0x7fffffffee18b : 0xff	0x7fffffffee193 : 0xff	0x7fffffffee19b : 0xff
0x7fffffffee184 : 0xff	0x7fffffffee18c : 0xff	0x7fffffffee194 : 0xff	0x7fffffffee19c : 0xff
0x7fffffffee185 : 0x7f	0x7fffffffee18d : 0x7f	0x7fffffffee195 : 0x7f	0x7fffffffee19d : 0x7f
0x7fffffffee186 : 0x00	0x7fffffffee18e : 0x00	0x7fffffffee196 : 0x00	0x7fffffffee19e : 0x00
0x7fffffffee187 : 0x00	0x7fffffffee18f : 0x00	0x7fffffffee197 : 0x00	0x7fffffffee19f : 0x00
p[0] = &a[0][0]	p[1] = &a[0][1]	p[3] = &a[1][0]	p[3] = &a[1][1]

1.5 结构与联合类型

最后，我们介绍**异构**（Heterogeneous）的数据类型，**结构**（`struct`）与**联合**（`union`）。正如我们刚才环形链表中所展示的，异构的数据类型，如 `struct`，可以容纳多种其他的基本数据类型。在**编译**的过程中，对一个 `struct` 不同成员（Field）的访问，都会被处理为对寄存器与内存的访问。因此，在 C 语言中，其实本质上我们是没有数据类型的，我们所有的只是 Byte。

`struct` 按照顺序，将它的成员（内存对齐地）按顺序保存在内存中；`union` 中的成员则共享低地址的内存，这是它们的主要区别。请参考下方的例子，注释中是从低地址到高地址，它们分布在内存上的值：

```
struct {
    uint32_t    field1;
    uint16_t    field2;
}   s;
s.field1 = 0xffffabcd;
s.field2 = 0x1234;
// 0xcd 0xab 0xff 0xff 0x34 0x12 0x00 0x00

union
{
    uint32_t    field1;
    uint16_t    field2;
}   u;
u.field1 = 0xffffabcd;
u.field2 = 0x1234;
// 0x34 0x12 0xff 0xff
```


对于这两类异构的数据类型，由于它们包容了许多种不同大小的数据类型，因此我们需要特别注意内存的**对齐**（Alignment）问题。关于对齐是这样的，我们只需要考虑**该数据类型的内存地址的低位零**即可。先前我们讨论过，内存的区分粒度是 Byte，因此两个相邻的地址在内存上总相隔一个 Byte。如果我们将内存看作从零地址 `0x0000,0000,0000,0000` 开始，只储存数据类型 `Type`，那么整个内存又可以看作关于类型 `Type` 的数组，而恰好在这数组上的 `Type` 元素（也即地址可以被数据类型的长度**整除**）便是被对齐了的。

不同长度的数据类型有不同的对齐方式，对齐方式主要展现在它们的低位零上，因为基本数据类型的长度总是 2 的幂次，因此低位 Bit 总是 `0`。我们可以对不同长度的数据类型给出下列表格：

基本数据类型	sizeof()	对齐后的 {0, 1} 地址
char, uint8_t	1	······,********
short, uint16_t	2	······,*******0
int, float, uint32_t	4	······,******00
(void *), double, uint64_t	8	······,*****000

union 按照 Byte 最长的数据类型占用内存即可，因为它的成员都是低地址对齐的，不会超过最长的数据类型。struct 同样按照最长的数据类型对齐它的初始地址，并且同样对齐它所占用的内存长度，同时，内部成员按照各自的对齐方式对齐自己的初始地址。因此，拥有同样成员的 struct 可以占用不同大小的内存，而我们可以将一个 struct 优化到最优，如下面的例子：

```
struct DATA_STRUCT {
  char c; long long l; float f; char s[3];
  int *i; int *j;
} data_t;
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| c |   |   |   |   |   |   |   | l | l | l | l | l | l | l | l |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| f | f | f | f | s | s | s |   | i | i | i | i | i | i | i | i |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| j | j | j | j | j | j | j | j |
+---+---+---+---+---+---+---+---+
```

而它有更紧致同时也等效的布局：

```
struct DATA_STRUCT {
  char c; char s[3]; float f;
  long long l; int *i; int *j;
} data_t;
```

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| c | s | s | s | f | f | f | f | l | l | l | l | l | l | l | l |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| i | i | i | i | i | i | i | i | j | j | j | j | j | j | j | j |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

•

Cutland 在可计算理论与递归函数的 [**cutland1980computability**] 中简略地介绍了 Godel 不完备定理的证明，并且给出了计算 Godel 配数的方法。信息论方面，Cover 的 [**cover1999elements**] 非常经典。尽管我们这一章只是简略地介绍了“信息来自于差异”这一观点，但读者仍可以去阅读这本书，从而了解更多关于信息论的观点。关于面向对象的许多工程思想，读者可以去阅读这一领域的经典著作，Booch 的 [**booch1990object**]。对于我们，更多的是思考如何在 C 语言这一面向过程的编程语言中实现面向对象，这一点其实是非常有趣的。CMU 本科生的教科书 [**bryant2003computer**] 深入地介绍了计算机系统，它的第二章也囊括了我们讨论的数据类型，关于整数加法方面，这本书有一幅精彩的三维图像。计算机上的整数

运算严格地构成了代数结构，读者可以参考 Rotman 的 [**rotman2000first**]，更加深入地了解群、环、域的概念。学习抽象代数的一大目标是为了解决经典的数学问题：判断高阶有理系数方程是否有代数解。当然这个远远超越了我们在这本书中要讨论的内容。在浮点数问题上，特别是近似问题，我们可以参考 Overton 所写的有关浮点数格式的 [**overton2001numerical**]。最后，计算机系统是建立在硬件的基础上的，有关硬件的，例如加法器溢出、Bool 代数化简，Wakerly 的 [**wakerly2008digital**] 有详细的介绍。