

在这两种情况下, `printf` 首先将这个字当作一个无符号数输出, 然后把它当作一个有符号数输出。以下是实际运行中的转换函数: $T2U_{32}(-1) = UMax_{32} = 2^{32} - 1$ 和 $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMin_{32}$ 。

由于 C 语言对同时包含有符号和无符号数表达式的这种处理方式, 出现了一些奇特的行为。当执行一个运算时, 如果它的一个运算数是有符号的而另一个是无符号的, 那么 C 语言会隐式地将有符号参数强制类型转换为无符号数, 并假设这两个数都是非负的, 来执行这个运算。就像我们将要看到的, 这种方法对于标准的算术运算来说并无多大差异, 但是对于像 `<` 和 `>` 这样的关系运算符来说, 它会导致非直观的结果。图 2-19 展示了一些关系表达式的示例以及它们得到的求值结果, 这里假设数据类型 `int` 表示为 32 位补码。考虑比较式 `-1 < 0U`。因为第二个运算数是无符号的, 第一个运算数就会被隐式地转换为无符号数, 因此表达式就等价于 `4294967295U < 0U` (回想 $T2U_w(-1) = UMax_w$), 这个答案显然是错的。其他那些示例也可以通过相似的分析来理解。

表 达 式	类 型	求 值
<code>0 == 0U</code>	无符号	1
<code>-1 < 0</code>	有符号	1
<code>-1 < 0U</code>	无符号	0*
<code>2147483647 > -2147483647-1</code>	有符号	1
<code>2147483647U > -2147483647-1</code>	无符号	0*
<code>2147483647 > (int) 2147483648U</code>	有符号	1*
<code>-1 > -2</code>	有符号	1
<code>(unsigned) -1 > -2</code>	无符号	1

图 2-19 C 语言的升级规则的效果

注: 非直观的情况标注了 ‘*’。当一个运算数是无符号的时候, 另一个运算数也被隐式强制转换为无符号。

将 $TMin_{32}$ 写为 `-2147483647-1` 的原因请参见网络旁注 DATA:TMIN。


 **练习题 2.21** 假设在采用补码运算的 32 位机器上对这些表达式求值, 按照图 2-19 的格式填写下表, 描述强制类型转换和关系运算的结果。

表 达 式	类 型	求 值
<code>-2147483647-1 == 2147483648U</code>		
<code>-2147483647-1 < 2147483647</code>		
<code>-2147483647-1U < 2147483647</code>		
<code>-2147483647-1 < -2147483647</code>		
<code>-2147483647-1U < -2147483647</code>		

网络旁注 DATA:TMIN C 语言中 $TMin$ 的写法

在图 2-19 和练习题 2.21 中, 我们很小心地将 $TMin_{32}$ 写成 `-2147483647-1`。为什么不简单地写成 `-2147483648` 或者 `0x80000000`? 看一下 C 头文件 `limits.h`, 注意到它们使用了跟我们写 $TMin_{32}$ 和 TMa_{32} 类似的方法:

```
/* Minimum and maximum values a 'signed int' can hold. */
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX - 1)
```

不幸的是, 补码表示的不对称性和 C 语言的转换规则之间奇怪的交互, 迫使我们用