In [3]:
```
use y2025assignment1: all;
```

Assignment: SaC Basics

Note that you can refine domain constraints in SaC by using a '|' symbol followed by a boolean expression after the function signature. For example, you can define:

In [4]:
```
double[n] recip (double[n] x)   |  all (x != 0.0)
{
    return 1.0/x;
}
```

In [5]:
```
recip ([2.0,0.0,3.0])
```

```
Dimension:  1
Shape    : <  3>
<5.000000e-01 inf 3.333333e-01 >
```

```
*** SAC runtime warning
*** In //var/folders/2t/gqvy03mn5jlgms6b0jffctt00000gp/T/jup-sacc51t41yo/t
mptj65995c.sac, line 13, column 1
*** Type pattern pre-condition of recip failed
```

myGenarray

Define your own version of `genarray` named `myGenarray`. Make sure that you capture *all* domain constraints in the signature of your function definition. Try to be as generic as possible. A few example applications of the original function `genarray` are:

In [6]:
```
myGenarray ([2], 42)
```

```
Dimension:  1
Shape    : <  2>
<42 42 >
```

In [7]:
```
myGenarray ([2], [1,2,3])
```

```
Dimension:  2
Shape    : <  2,  3>
| 1  2  3 |
| 1  2  3 |
```

In [8]:
```
myGenarray ([2,2], [1,2,3,4])
```

```
Dimension:  3
Shape    : <  2,  2,  4>
< 1  2  3  4 > < 1  2  3  4 >
< 1  2  3  4 > < 1  2  3  4 >
```

In [9]:
```
myGenarray ([], [1,2,3,4])
```

```
Dimension:  1
Shape    : <  4>
< 1  2  3  4 >
```

genarrayI

Now try to define a function `genarrayI` which also constructs arrays but which replicates elements on the "inside". Again, please do make `genarrayI` as generic as possible and do capture all domain constraints within the function definition. Example applications are:

In [8]:
```
genarrayI ([2], 42)
```

```
Dimension:  1
Shape    : <  2>
<42 42 >
```

In [9]:
```
genarrayI ([2], [1,2,3])
```

```
Dimension:  2
Shape    : <  3,  2>
| 1  1 |
| 2  2 |
| 3  3 |
```

In [10]:
```
genarrayI ([2,2], [1,2,3,4])
```

```
Dimension:  3
Shape    : <  4,  2,  2>
< 1  1 > < 1  1 >
< 2  2 > < 2  2 >
< 3  3 > < 3  3 >
< 4  4 > < 4  4 >
```

In [11]:
```
genarrayI ([], [1,2,3,4])
```

```
Dimension:  1
Shape    : <  4>
< 1  2  3  4 >
```

myTake1

Re-define the function `take` as a new function `myTake1`. Restrict the function's domain so that the length of the first argument must not exceed the dimensionality of the second argument, and that all elements of the first argument constitute a prefix of a legal index into the second. Eexample applications are:

In [12]:
```
myTake1 ([1], [21,42])
```

```
Dimension:  1
Shape    : <  1>
<21 >
```

In [13]:
```
myTake1 ([2,3], reshape ([10,5], iota(50)))
```

```
Dimension:  2
Shape    : <  2,  3>
| 0  1  2 |
| 5  6  7 |
```

In [14]:
```
myTake1 ([1], reshape ([10,5], iota(50)))
```

```
Dimension:  2
Shape    : <  1,  5>
| 0  1  2  3  4 |
```

In [15]:
```
myTake1 ([], reshape ([10,5], iota(50)))
```

```
Dimension:  2
Shape    : < 10,  5>
| 0  1  2  3  4 |
| 5  6  7  8  9 |
|10 11 12 13 14 |
|15 16 17 18 19 |
|20 21 22 23 24 |
|25 26 27 28 29 |
|30 31 32 33 34 |
|35 36 37 38 39 |
|40 41 42 43 44 |
|45 46 47 48 49 |
```

In [16]:
```
myTake1 ([0], reshape ([10,5], iota(50)))
```

```
Dimension:  2
Shape    : <  0,  5>
<>
```

myTake2

Define a more generic variant of `take` named `myTake2`. In contrast to `myTake1`, it should allow for the first argument's values to be bigger than the corresponding

shape components of the second argument. For "missing" elements in the second argument, we insert the value `0`. Example applications are:

In [17]:
```
myTake2 ([2], [1])
```

```
Dimension:  1
Shape    : <  2>
< 1  0 >
```

In [18]:
```
myTake2 ([11,6], reshape ([10,5], iota(50)))
```

```
Dimension:  2
Shape    : < 11,  6>
| 0  1  2  3  4  0 |
| 5  6  7  8  9  0 |
|10 11 12 13 14  0 |
|15 16 17 18 19  0 |
|20 21 22 23 24  0 |
|25 26 27 28 29  0 |
|30 31 32 33 34  0 |
|35 36 37 38 39  0 |
|40 41 42 43 44  0 |
|45 46 47 48 49  0 |
| 0  0  0  0  0  0 |
```

myTake3

Try to define a final version of `take` which allows for the first argument to have more elements than the dimensionality of the second argument. Can you come up with a consistent definition?