Functions

- Functions form a special class of relations that satisfy additional requirement: any element of the source set can be related to no more than 1 element of the target
- Functionality requirement mathematically: $(x, y) \in R \wedge (x, z) \in R \;\Rightarrow\; y = z$
- Any operation applicable to a relation or a set is also applicable to a function. For example, we can talk about the domain and the range of a function.
- If f is a function, then $f(x)$ is the result of the function f for the argument x

Functions (cont.)

- Functions are called *total* if their domain is the whole source set.
- Functions are called *partial* if their domain is a subset of the source set.
- Functions are called *injective* or (*one-to-one*) if for every element y from their range exists only one element x from their domain such that $f(x) = y$.
- Functions are called *surjective* if their range is the whole target set.

Varieties of functions

Suppose we have a function f (from the source X to the target Y). Then it is called

Total function	$\rightarrow$	`-->`	if $dom(f) = X,\ ran(f) \subseteq Y$
Partial function	$\pfun$	`+->`	if $dom(f) \subseteq X,\ ran(f) \subseteq Y$
Total injection	$\rightarrowtail$	`>->`	if $dom(f) = X,\ ran(f) \subseteq Y$ and one-to-one function
Partial injection	$\pinj$	`>+>`	if $dom(f) \subseteq X,\ ran(f) \subseteq Y$ and one-to-one function
Total surjection	$\twoheadrightarrow$	`-->>`	if $dom(f) = X,\ ran(f) = Y$
Partial surjection	$\psurj$	`+->>`	if $dom(f) \subseteq X,\ ran(f) = Y$
Bijection	$\bij$	`>->>`	if $dom(f) = X,\ ran(f) = Y$ and one-to-one function

Lambda notation for functions

- In addition to defining functions as sets of pairs (relations), *lambda notation* can be used to introduce new functions.
- Lambda notation allows us to define a new function f by describing the result $f(x)$ for any given argument x.
- The general form of a function is then

  $$\lambda x \bullet (x \in T \mid E)$$

  "the function maps x, of type T, to the value E".
- The corresponding ASCII notation – %x. (x:T | E)

Sequences

- Sequences are used to describe finite ordered lists of elements of a given type.
- A sequence over a set S is a total function from an interval $1..n$ (for some $n \in NAT$) to S.
- In a sequence, elements are ordered and may appear more than once.
- Any operation applicable to a function, a relation, or a set is also applicable to a sequence.

Operations on Sequences

$[e_1, \cdots, e_n]$	The sequence containing elements $e_1, \cdots, e_n$ This is the same as $\{(1, e_1), \cdots, (n, e_n)\}$
[e]	The singleton sequence with element e
[] $(<>)$	The empty sequence
$seq(S)$	The set of finite sequences of elements from S
$seq1(S)$	The set of finite non-empty sequences of elements from S: $seq1(S) = seq(S) - \{[]\}$
$iseq(S)$	The set of injective sequences of elements from S, i.e. sequences without repetitions
$perm(S)$	Permutations of elements from a finite S, i.e. sequences that contain all elements from S without repetitions

Operations on Sequences (cont.)

$size(s)$	The size of sequence s
$rev(s)$	The reverse of s
$first(s)$	The first element of non-empty s
$last(s)$	The last element of non-empty s
$tail(s)$	The sequence s, with its first element removed (for non-empty s)
$front(s)$	The sequence s, with its last element removed (for non-empty s)
$s \uparrow n$ $(s/\|\backslash n)$	The sequence s with the first n elements retained, $n \leq size(s)$
$s \downarrow n$ $(s\backslash\|/n)$	The sequence s with the first n elements removed, $n \leq size(s)$

Operations on Sequences (cont.)

$s \,^\wedge\, t$	The concatenation of sequences s and t
$e \rightarrow s$ $(e -> s)$	The sequence formed by prepending e to s
$s \leftarrow e$ $(s <- e)$	The sequence formed by appending e to s
$``normal\ text''$	A sequence of characters

Arrays

- An array is a named, indexed collection of values of a given type.
- The array values can be accessed (read and updated) by using appropriate indexes.
- If we use $1..n$ (for some $n \in NAT$) as our index set, then an array (of type S) can be modelled as a sequence from $1..n$ to S.
- In fact, any set can be used as the index set for arrays. Therefore, arrays can be modelled as total functions from S (index set) to T (the type of array values).

Array assignment

- The abstract machine notation allows us to assign values to indexed elements of arrays:

$$a(i) \ := \ E$$

- This is the shorthand for the following assignment:

$$a \ := \ a <\!\!+ \ \{(i, E)\}$$

- The weakest precondition for the array assignment is calculated then as follows

$$[a(i) \ := \ E]\, P \ = \ P[a <\!\!+ \{(i, E)\} \ / \ a]$$

```
MACHINE Hotel(sze)
CONSTRAINTS sze ∈ NAT1

SETS ROOM
CONSTANTS single, double
PROPERTIES
   card(ROOM) = sze ∧
   single ⊆ ROOM ∧
   double ⊆ ROOM ∧
   single ∩ double = {} ∧
   single ∪ double = ROOM

VARIABLES guests
INVARIANT
   guests ∈ ROOM → 0..2 ∧
   guests[single] ⊆ 0..1 ∧
   guests[double] ⊆ 0..2
INITIALISATION
   guests := ROOM × {0}

OPERATIONS
   ...
```



```
  ...
  checkin(rr,nn) =
    PRE rr ∈ ROOM ∧ nn ∈ 1..2 ∧
        guests(rr) = 0 ∧
        (rr ∈ single ⇒ nn = 1)
    THEN guests(rr) := nn
    END;

  checkout(rr) =
    PRE rr ∈ ROOM
    THEN guests(rr) := 0
    END;

  change_room(rr1,rr2) =
    PRE
        rr1 ∈ ROOM ∧ rr2 ∈ ROOM ∧
        rr1 ≠ rr2 ∧ guests(rr1) > 0 ∧
        guests(rr2) = 0 ∧
        (rr2 ∈ single ⇒ guests(rr1) = 1)
    THEN
      guests := guests <+
        {(rr1,0), (rr2,guests(rr1))}
    END;
```

```
  ...
  nn ← roomquery(rr) =
    PRE rr ∈ ROOM
    THEN nn := guests(rr)
    END;

  nn ← vacancies =
    BEGIN
      nn := card (guests |> {0})
    END;
END
```