1. Introduction

The built-in library of XE contains a rather small set of abstractions. There exists a number of other abstractions that are often needed, but not included in the built-in library. The *utility library* contains a set of such useful abstractions – data types, procedures, and iterators.

The following sections describe these abstractions. The description of each type includes a short overview of the abstraction represented by it and the operations that the type has. The header of each operation is given in the normal XE syntax including the possible restrictions (**where** clause). The normal effects of an operation are listed in item **description**. If the operation can terminate exceptionally, the condition for each signal and the result objects are listed in item **signals**. If some operation can not be executed, for example due to lack of memory space, **failure** is signaled.

Some routines (operations) have side-effects, i.e., modify their argument objects. These routines have item **modifies** that lists the arguments modified. Not all of the routines with item **modifies** always modify their arguments, because some side-effects are secondary. A Secondary side-effect exists when the operations in the **where** clause of a routine can modify the arguments, but the routine itself does not. Library routines are described with the same notation as data type operations.

If an operation can be invoked using the syntactic sugar of XE then the sugared form of invocation is given in item **sugared**. All utility library types have the `dump` and `undump` operations described in the XE Rerence Manual.

2. Utility Types

2.1 Utility Type Generator ASSMEM

For each pair of types I and E, there exists a type `assmem[I, E]`, if the type I has operations

gt, lt, equal, similar: **proctype**(I, I) **returns**(bool)

These operations should have no side-effects. Objects of types `assmem[I, E]` are mutable and they model associative memories.

The abstraction of an associative memory is like a partial function. A partial function f: I $\rightarrow$ E is a subset of cartesian product I $\times$ E, a set containing pairs like (x y), where x is a member of the domain and y a member of the range. For an `assmem[I, E]` the domain is the set of all the values of the index type I and the range is set of all the values of the element type E. It is important to notice that given an x object the assmem can return the y object of the pair, but the inverse operation is not possible. The first element of a pair (x) is called the *label* and the second element (y) is called the *item*.

Operations

new = **proc**() **returns**(assmem[I, E])

Description Returns a new empty assmem.

inverse = **proc**(a: assmem[I, E]) **returns**(assmem[I, E])
where E **has** gt, lt, equal, similar: **proctype**(E, E) **returns**(bool)

Description Creates an inverse of the given assmem. This means that an assmem where labels are items and items are labels is returned. Therefore the result is an inverse of the partial function represented by the original assmem.

size = **proc**(a: assmem[I, E]) **returns**(int)

Description Returns number of pairs defined.

empty = **proc**(a: assmem[I, E]) **returns**(bool)

Description Returns **true** if size(a) = 0.

fetch = **proc**(a: assmem[I, E], label: I) **returns**(E) **signals**(not_found)

Description Returns the item associated with the given *label*.
Signals not_found when *label* is not in the assmem.
Sugared form of invocation is a[label]

store = **proc**(a: assmem[I, E], label: I, item: E)

Description Associates *label* to *item*. If *label* is already associated to an item, the item is replaced (adds a new pair or replaces the old one).
Modifies state of a.
Sugared form of invocation is a[label] := item

remove = **proc**(a: assmem[I, E], label: I) **returns**(E) **signals**(not_found)

Description Removes *label* and the associated item from the assmem.
Modifies state of a.
Signals not_found when label is not in assmem.

least = **proc**(a: assmem[I, E]) **returns**(I) **signals**(not_found)

Description Returns smallest label in a.
Signals not_found when empty(a).

greatest = **proc**(a: assmem[I, E]) **returns**(I) **signals**(not_found)

Description Returns biggest label in a.
Signals not_found when empty(a).

legal_index = **proc**(a: assmem[I, E], label: I) **returns**(bool)

Description Returns **true** if *label* is in the assmem.

from_to = **iter**(a: assmem[I, E], low, high: I) **yields**(I)

Description Iterates over a range of labels (increasing order). Yields all labels $\{x \mid low \leq x \leq high\}$.

from_downto = **iter**(a: assmem[I, E], high, low: I) **yields**(I)

Description Iterates over a range of labels (decreasing order). Yields all labels $\{x \mid low \leq x \leq high\}$.

elements = **iter**(a: assmem[I, E]) **yields**(E)

Description Iterates over all items stored in increasing order of their labels.

elements_down = **iter**(a: assmem[I, E]) **yields**(E)

Description Iterates over all items stored in decreasing order of their labels.

indexes = **iter**(a: assmem[I, E]) **yields**(I)

Description Iterates over labels (increasing order).

indexes_down = **iter**(a: assmem[I, E]) **yields**(I)

Description Iterates over labels (decreasing order).

pairs = **iter**(a: assmem[I, E]) **yields**(I, E)

Description Iterates over label–item pairs (increasing order).

pairs_down = **iter**(a: assmem[I, E]) **yields**(I, E)

Description Iterates over label–item pairs (decreasing order).

equal = **proc**(a, b: assmem[I, E]) **returns**(bool)

Description Returns **true** if the arguments are the same object.
Sugared form of invocation is a = b

similar = **proc**(a, b: assmem[I, E]) **returns**(bool)
where I **has** similar: **proctype**(I, I) **returns**(bool)
where E **has** similar: **proctype**(E, E) **returns**(bool)

Description Returns **true** if the arguments have similar labels associated with similar items.
Modifies can have a secondary side-effect on the arguments due to I$similar and E$similar.

equal_state = **proc**(a, b: assmem[I, E]) **returns**(bool)

Description Returns **true** if the arguments have equal labels associated with equal items.
Modifies can have a secondary side-effect on the arguments due to I$equal and E$equal.

copy = **proc**(a: assmem[I, E]) **returns**(assmem[I, E])
where I **has** copy: **proctype**(I) **returns**(I)
where E **has** copy: **proctype**(E) **returns**(E)

Description Returns a copy of *a*.
Modifies can have a secondary side-effect on the argument due to I$copy and E$copy.

copy_state = **proc**(a: assmem[I, E]) **returns**(assmem[I, E])

Description Returns a new assmem that has the same state as a.

parse = **proc**(s: string) **returns**(assmem[I, E]) **signals**(syntax_error(string, int))
where I **has** parse: **proctype**(string) **returns**(I) **signals**(*)
where E **has** parse: **proctype**(string) **returns**(E) **signals**(*)

Description Parses a string of the form "`[` label1: item1, ... `]`" to an assmem with similar contents.
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(a: assmem[I, E]) **returns**(string)
where I **has** unparse: **proctype**(I) **returns**(string)
where E **has** unparse: **proctype**(E) **returns**(string)

Description Returns a string of the form "`[` label1: item1, ... `]`" representing a.
Modifies can have a secondary side-effect on the argument due to I$unparse and E$unparse.

2.2 Utility Type BIGINT

Objects of the type `bigint` are immutable and model mathematical integers. The size of the integers is limited only by the maximum object size of the system. (So, `bigints` can be quite big). If the maximum size is exceeded, **failure** is signalled. Objects of the type `bigint` use the integer literal form described in the XE Reference Manual.

Operations

create = **proc**(i: sequence[byte]) **returns**(bigint)

Description Returns the `bigint` corresponding to the unsigned integer value of i: $2^{0*bytesize} * s_k + 2^{1*bytesize} * s_{k-1} + ...$, where k is the last index of the sequence .
Sugared form of invocation is the integer literal.

succ = **proc**(i: bigint) **returns**(bigint)

Description Returns the successor of i.

pred = **proc**(i: bigint) **returns**(bigint)

Description Returns the predecessor of i.

add = **proc**(x, y: bigint) **returns**(bigint)

Description Integer addition.
Sugared form of invocation is x + y.

sub = **proc**(x, y: bigint) **returns**(bigint)

Description Integer subtraction.
Sugared form of invocation is x - y.

minus = **proc**(x: bigint) **returns**(bigint)

Description Returns the negative of x.
Sugared form of invocation is - x.

mul = **proc**(x, y: bigint) **returns**(bigint)

Description Integer multiplication.
Sugared form of invocation is x * y.

div = **proc**(x, y: bigint) **returns**(bigint) **signals**(zero_divide)

Description Returns the integer quotient of dividing x by y.
Signals zero_divide if $y = 0$.
Sugared form of invocation is x / y.

mod = **proc**(x, y: bigint) **returns**(bigint) **signals**(zero_divide)

Description Returns the remainder of dividing x by y.
Signals zero_divide if $y = 0$.
Sugared form of invocation is x // y.

abs = **proc**(x: bigint) **returns**(bigint)

Description Returns the absolute value of x.

power = **proc**(x, y: bigint) **returns**(bigint) **signals**(negative_exponent)

Description Returns x to the power y.
Signals negative_exponent if $y < 0$.
Sugared form of invocation is x ^ y.

from_to_by = **iter**(f, t, b: bigint) **yields**(bigint)

Description Yields the integers from f to t with increment b, that is, $f, f + b, ..., f + n * b$, where n is the largest integer such that $f + n * b \leq t$ if b is positive and $f + n * b \geq t$ otherwise. Terminates immediately if no such non-negative n exists. If $b = 0$, loops.

from_to = **iter**(f, t: bigint) **yields**(bigint)

Description Yields all integers from f to t.

from_downto = **iter**(f, t: bigint) **yields**(bigint)

Description Yields all integers from f down to t.

lt = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if x is less than y.
Sugared form of invocation is x `<` y.

le = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if x is less than or equal to y.
Sugared form of invocation is x `<=` y.

ge = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if x is greater than or equal to y.
Sugared form of invocation is x `>=` y.

gt = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if x is greater than y.
Sugared form of invocation is x `>` y.

equal = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if $x = y$.
Sugared form of invocation is x `=` y.

similar = **proc**(x, y: bigint) **returns**(bool)

Description Returns **true** if $x = y$.

copy = **proc**(x: bigint) **returns**(bigint)

Description Returns its argument.

to_int = **proc**(x: bigint) **returns**(int) **signals**(overflow)

Description Returns the `int` having the same value as x.
Signals overflow if result would lie outside of the allowed range.

from_int = **proc**(x: int) **returns**(bigint)

Description Returns the `bigint` having the same value as x.

parse = **proc**(s: string) **returns**(bigint) **signals**(bad_format)

Description Converts string s representing an integer literal and returns the corresponding integer.
Signals bad_format if s is not a legal integer literal.

unparse = **proc**(x: bigint) **returns**(string)

Description Produces a decimal integer literal corresponding to x.

2.3 Utility Type BITVECTOR

Objects of type `bitvector` are immutable and they model bit vectors of arbitrary size. They can also be treated as positive integers (i.e. unsigned integers). Integer value of a bitvector $b = \langle b_h, ..., b_0 \rangle$ is $\sum_{i=0}^{h} 2^i b_i$. Thus the least significant bit is the right most bit and has the index 0. The normal integer literal can be used to create `bitvector` objects. The leading (right most) zeros are omitted, i.e. the right most bit (if it exists) is always one.

Operations

create = **proc**(i: sequence[byte]) **returns**(bitvector)

Description Returns the bitvector given in the sequence i.
Sugared form of invocation is the integer literal.

size = **proc**(x: bitvector) **returns**(bitvector)

Description Returns the number of bits in x.

bitand = **proc**(x, y: bitvector) **returns**(bitvector)

Description Bitwise and-operation of x and y. If x and y are of different length, the non-existing bits at left end are treated as zeros.
Sugared form of invocation is x `&` y.

bitor = **proc**(x, y: bitvector) **returns**(bitvector)

Description Bitwise or-operation of x and y. If x and y are of different length, the non-existing bits at left end are treated as zeros.
Sugared form of invocation is x `|` y.

bitxor = **proc**(x, y: bitvector) **returns**(bitvector)

Description Bitwise exclusive or-operation of x and y. If x and y are of different length, the non-existing bits at left end are treated as zeros.

bitnot = **proc**(x: bitvector) **returns**(bitvector)

Description	Bitwise complement of x.
Sugared	form of invocation is ~x.

shift_left = **proc**(x: bitvector, n: int) **returns**(bitvector)

Description	Returns the bitvector where the bits of x are shifted left n positions.

shift_right = **proc**(x: bitvector, n: int) **returns**(bitvector)

Description	Returns the bitvector where the bits of x are shifted right n positions. The n rightmost bits are shifted "out".

rotate_left = **proc**(x: bitvector, n: int) **returns**(bitvector)

Description	Returns the bitvector where the bits of x are rotated left n positions.

rotate_right = **proc**(x: bitvector, n: int) **returns**(bitvector)

Description	Returns the bitvector where the bits of x are rotated right n positions.

fetch = **proc**(x: bitvector, i: int) **returns**(bool) **signals**(bounds)

Description	Returns **true** if the i^{th} bit of x is one. The least significant bit has index 0.
Signals	bounds if $i \notin \{0, ..., size_x - 1\}$.
Sugared	form of invocation is `x[i]`.

replace = **proc**(x: bitvector, i: int, b: bool) **returns**(bitvector) **signals**(bounds)

Description	Returns a copy of x, where the i^{th} bit is set to one if b is **true**, otherwise to zero. The least significant bit has index 0.
Signals	bounds if $i < 0$.

equal = **proc**(x, y: bitvector) **returns**(bool)

Description	Returns **true** if $x = y$.
Sugared	form of invocation is x `=` y.

similar = **proc**(x, y: bitvector) **returns**(bool)

Description	Returns **true** if $x = y$.

copy = **proc**(x: bitvector) **returns**(bitvector)

Description	Returns its argument.

to_word = **proc**(x: bitvector) **returns**(word) **signals**(overflow)

Description	Returns the word having the same value as x.
Signals	overflow if the result would lie outside of the allowed range.

from_word = **proc**(x: word) **returns**(bitvector)

Description	Returns the bitvector having the same value as x.

to_bigint = **proc**(x: bitvector) **returns**(bigint)

Description	Returns the bigint having the same integer value as x.

from_bigint = **proc**(x: bigint) **returns**(bitvector) **signals**(overflow)

Description	Returns the bitvector having the same integer value as x.
Signals	overflow if $x < 0$.

parse = **proc**(s: string) **returns**(bitvector) **signals**(bad_format)

Description	Converts string s representing an integer literal and returns the corresponding bitvector.
Signals	bad_format if s is not a legal integer literal.

unparse = **proc**(b: bitvector) **returns**(string)

Description	Produces a hexadecimal integer literal corresponding to b.

2.4 Utility Type DATE

Objects of type `date` are immutable and represent calendar dates and times. Years before Christ cannot be represented. Dates can be created using the struct literal.

Operations

Besides the operations below `date` has the usual `dump` and `undump` operations.

create = **proc**(d: struct{day,month,year,hour,minute,second,millis,micros: int})
returns(date) **signals**(bad_component)

Description	Creates a new date.
Signals	bad_component if some of the components is not in the allowed range.
Sugared	form of invocation is the struct literal.

sub = **proc**(d1, d2: date) **returns**(time) **signals**(negative_time)

Description	Returns the amount of time between from $d2$ to $d1$.
Signals	negative_time, if $d1$ is before $d2$.

incr = **proc**(d: date, t: time) **returns**(date)

Description	Increments d by t.

decr = **proc**(d: date, t: time) **returns**(date) **signals**(before_christ)

Description	Decrements d by t.
Signals	before_christ if the date would earlier than 1-Jan-1.

get_day = **proc**(d: date) **returns**(int)

Description	Returns the number of the day.
Sugared	form of invocation is d.day

get_month = **proc**(d: date) **returns**(int)

Description	Returns the number of the month.
Sugared	form of invocation is d.month

get_year = **proc**(d: date) **returns**(int)

Description	Returns the year.
Sugared	form of invocation is d.year

get_hour = **proc**(d: date) **returns**(int)

Description	Returns the hour.
Sugared	form of invocation is d.hour

get_minute = **proc**(d: date) **returns**(int)

Description	Returns the hour.
Sugared	form of invocation is d.minute

get_second = **proc**(d: date) **returns**(int)

Description	Returns the hour.
Sugared	form of invocation is d.second

get_millis = **proc**(d: date) **returns**(int)

Description	Returns the milliseconds.
Sugared	form of invocation is d.millis

get_micros = **proc**(d: date) **returns**(int)

Description Returns the microseconds.
Sugared form of invocation is d.micros

lt = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is before $d2$.

le = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is before, or the same as $d2$.

ge = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is after, or the same as $d2$.

gt = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is after $d2$.

equal = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is the same date as $d2$.
Sugared form of invocation is d1 = d2

similar = **proc**(d1, d2: date) **returns**(bool)

Description Returns **true** if $d1$ is the same date as $d2$.

copy = **proc**(d: date) **returns**(date)

Description Returns its argument.

to_struct = **proc**(d: date)
returns(struct{day,month,year,hour,minute,second,millis,micros: int })

Description Returns the date represented as a struct.

parse = **proc**(s: string) **returns**(date) **signals**(syntax_error(string, int))

Description Parses a string like "11-May-1988 11:49:15.6713". The name of the month can be abbreviated (the abbreviation must define a unique prefix). The case of the letters is ignored. The time (or any rightmost component of it) can be left out, e.g. "11-Nov-1988" means the same as "11-November-1988 00:00:00".
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(d: date) **returns**(string)

Description Returns the date with day time, e.g., "11-November-1988 11:52:17".

unparse_date = **proc**(d: date) **returns**(string)

Description Returns the date, e.g., "11-November-1988".

unparse_time = **proc**(d: date) **returns**(string)

Description Returns the day time component of the date in the form "11:55:49".

2.5 Utility Type Generator FIFO

For each type T there exists type `fifo[T]`. Objects of a type `fifo[T]` are mutable and they model first-in-first-out queues of the element type T. Elements of a queue are indexed incrementally, so that the first element has index 0.

new = **proc**() **returns**(fifo[T])

Description Returns a new empty queue.

create = **proc**(f: sequence[t]) **returns**(fifo[T])

Description Returns a new queue that has the same elements as s (indexes remain).
Sugared form of invocation is the sequence constructor.

size = **proc**(f: fifo[T]) **returns**(int)

Description Returns number of elements in the queue.

empty = **proc**(f: fifo[T]) **returns**(bool)

Description Returns **true** if f is empty.

get = **proc**(f: fifo[T]) **returns**(T) **signals**(empty)

Signals empty if there are no elements in f.

getn = **proc**(f: fifo[T], n: int) **returns**(sequence[T]) **signals**(rest(sequence[T]))

Description Returns the n first elements of the queue (order remains).
Signals rest(s) if $size_f < n$, where s contains the remaining elements.

put = **proc**(f: fifo[T], e: t)

Description Puts a new element to the end of the queue.

putn = **proc**(f: fifo[T], s: sequence[T])

Description Puts the elements in s to the end of the queue (in their increasing order of indexes).

peek = **proc**(f: fifo[T]) **returns**(T) **signals**(empty)

Description Returns the first element in the queue.
Signals empty if $size_f = 0$.

fetch = **proc**(f: fifo[T], i: int) **returns**(T) **signals**(bounds)

Description Returns i_{th} (starting from 0) element from the queue.
Signals bounds if i is not a legal index.
Sugared form of invocation is `f[i]`

elements = **iter**(f: fifo[T]) **yields**(T)

Description Yields elements of the queue in their queue order.

indexes = **iter**(f: fifo[T]) **yields**(int)

Description Yields integers 0 .. $size_f - 1$.

equal = **proc**(s1, s2: fifo[T]) **returns**(bool)

Description Returns **true** if $s1$ and $s2$ are the same queue.
Sugared form of invocation is s1 `=` s2

similar = **proc**(s1, s2: fifo[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if the queues are similar.

equal_state = **proc**(s1, s2: fifo[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if the queues have equal states.

copy = **proc**(f: fifo[T]) **returns**(T)
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of f.

copy_state = **proc**(f: fifo[T]) **returns**(fifo[T])

Description Returns a fifo that has the same state as f.

parse = **proc**(s: string) **returns**(fifo[T]) **signals**(syntax_error(string, int))
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parse a string of the form "[element, ...]".
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(f: fifo[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string of the form "[element, ...]".

2.6 Utility Type Generator FLAG

Flags are like tags, but they are mutable. (Tags are described in the XE Reference Manual.)

Operations

create_N_i = **proc**(n: null) **returns**(flag{F})

Description Returns the flag N_i. There is a create_N_i operation for each flag.
Sugared form of invocation is 'N_i

is_N_i = **proc**(f: flag{F}) **returns**(bool)

Description Returns **true** if the flag f is N_i. There is an is_N_i operation for each flag.

set_N_i = **proc**(f: flag{F})

Description Sets f to be N_i.

tag_of = **proc**(f: flag{F}) **returns**(tag{F})

Description Returns its argument (for the **tagcase** statement).

equal = **proc**(f1, f2: flag{F}) **returns**(bool)

Description Returns **true** if $f1 = f2$.
Sugared form of invocation is f1 = f2.

similar = **proc**(f1, f2: flag{F}) **returns**(bool)

Description Returns **true** if $f1 = f2$.

copy = **proc**(f: flag{F}) **returns**(flag{F})

Description Returns its argument.

from_tag = **proc**(t: tag{F}) **returns**(flag{F})

Description Returns the flag corresponding to tag t.

parse = **proc**(s: string) **returns**(flag{F}) **signals**(bad_format)

Description Returns the flag object corresponding to the given string.
Signals bad_format if s does not match any N_i.

unparse = **proc**(f: flag{F}) **returns**(string)

Description Returns a string representing flag f.

2.7 Utility Type Generator LIST

For each type T there exists a type `list[T]`. Objects of type `list[T]` represent dynamic n-tuples that can shrink and grow in size. Elements of a list L are indexes from 0 to $size_L - 1$ ($= high_L$).

new = **proc**() **returns**(list[T])

Description Returns a new empty list.

create = **proc**(l: sequence[T]) **returns**(list[T])

Description Creates a new list $\langle l_0, ..., l_{high} \rangle$.
Sugared form of invocation is the sequence constructor.

fill = **proc**(n: int, e: T) **returns**(list[T]) **signals**(negative_size)

Description Returns a new list of size n filled with element e.
Signals Signals negative_size if $n < 0$.

fill_copy = **proc**(n: int, e: T) **returns**(list[T]) **signals**(negative_size)
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a new list of size n filled with copies of element e.
Signals Signals negative_size if $n < 0$.

size = **proc**(l: list[T]) **returns**(int)

Description Return size of l.

cut = **proc**(l: list[T], lb, n: int) **signals**(bounds, negative_size)

Description Modifies the list to contain the elements $\langle l_{lb}, ..., l_{lb+n-1} \rangle$, or an empty list $\langle\rangle$ if $n = 0$.
Modifies state of l.
Signals negative_size if $n < 0$. If $lb < 0$ or $lb + n - 1 > size_l$ signals bounds.

part = **proc**(l: list[T], lb, n: int) **returns**(list[T]) **signals**(bounds, negative_size)

Description Returns a new list $\langle l_{lb}, ..., l_{lb+n-1} \rangle$, or an empty list $\langle\rangle$ if $n = 0$.
Signals negative_size if $n < 0$. If $lb < 0$ or $lb + n - 1 > size_l$ signals bounds.

split = **proc**(l: list[T], pos: int) **returns**(list[T], list[T]) **signals**(bounds)

Description Returns two new lists $\langle l_0, ..., l_{n-1} \rangle$ and $\langle l_n, ..., l_{high} \rangle$.
Signals bounds if $pos < 0$ or $pos > size_l$.

concat = **proc**(l1, l2: list[T]) **returns**(list[T])

Description Returns a new list $\langle l1_0, ..., l1_{high_{s1}}, l2_0, ..., l2_{high_{l2}} \rangle$.
Sugared form of invocation is l1 || sl.

fetch = **proc**(l: list[T], i: int) **returns**(T) **signals**(bounds)

Description Returns the element l_i of l.
Signals bounds if $i \notin \{0, ..., high_l\}$.
Sugared form of invocation is `l[i]`.

store = **proc**(l: list[T], i: int, x: t) **signals**(bounds)

Description Replaces the element l_i of l with x.
Modifies state of l.
Signals bounds if $i \notin \{0, ..., high_l\}$.
Sugared form of invocation is `l[i]` := x.

top = **proc**(l: list[T]) **returns**(T) **signals**(empty)

Description Returns the topmost element of l, l_{high}.
Signals empty if l is empty.

insert = **proc**(l: list[T], i: int, x: t) **signals**(bounds)

Description Increases size of the list by one. Shift elements $l_i, ..., l_{high}$ one position higher and stores x as l_i.
Modifies state of l.
Signals bounds if $i \notin \{0, ..., size_l\}$.

remove = **proc**(l: list[T], i: int) **returns**(T) **signals**(bounds)

Description Decreases size of the list by one. Removes element l_i and shift elements $l_{i+1}, ..., l_{high}$ one position lower. Returns the removed element.
Modifies state of l.
Signals bounds if $i \notin \{0, ..., high_l\}$.

insert_top = **proc**(l: list[T], x: T)

Description Inserts a new element to the high end of the list.

remove_top = **proc**(l: list[T], x: T) **returns**(T) **signals**(empty)

Description Removes the top most element and returns it.
Signals empty if the list is empty.

map = **proc**(l: list[T], f: **proctype**(t) **returns**(t) **signals**(*))
returns(list[T]) **signals**(not_possible(string))

Description Returns a new list $\langle f(l_0), ..., f(l_{high})\rangle$.
Signals not_possible(x), if any invocation of f signals; x is the name of the signal.

elements = **iter**(l: list[T]) **yields**(T)

Description Yields all elements l_i in increasing order of indexes.

elements_down = **iter**(l: list[T]) **yields**(T)

Description Yields all elements l_i in decreasing order of indexes.

indexes = **iter**(l: list[T]) **yields**(int)

Description Yields all indexes of a list in increasing order.

indexes_down = **iter**(l: list[T]) **yields**(int)

Description Yields all indexes of a list in decreasing order.

equal = **proc**(l1, l2: list[T]) **returns**(bool)

Description Returns **true** if both arguments are the same object.
Sugared form of invocation is l1 = l2.

similar = **proc**(l1, l2: list[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if t$similar($l1_i, l2_i$) is **true** for all indexes i.
Modifies can have a secondary side-effect on $l1$ and $l2$ due to t$similar.

equal_state = **proc**(l1, l2: list[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if $l1$ and $l2$ have the same state.
Modifies can have a secondary side-effect on $l1$ and $l2$ due to t$equal.

copy = **proc**(l: list[T]) **returns**(list[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of l, $\langle e_0, ..., e_{high_l} \rangle$, where e_i = t$copy(l_i) for all indexes i.
Modifies can have a secondary side-effect on l due to t$copy.

copy_state = **proc**(l: list[T]) **returns**(list[T])

Description Creates a new list with the same state as l.

to_sequence = **proc**(l: list[T]) **returns**(sequence[T])

Description Returns the sequence that has the same elements as l.

from_array = **proc**(a: array[T]) **returns**(list[T])

Description Returns a list that has the same elements as a.

to_array = **proc**(l: list[T]) **returns**(array[T])

Description Returns an array that has the same elements as l.

parse = **proc**(s: string) **returns**(list[T]) **signals**(syntax_error(string, int))
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a list structure of the syntax [*element* , ...] where each list *element* is represented in the format of t$parse. Returns a new list corresponding to the string.
Signals syntax_error(m, i) where m is the error message and i the position in s.

unparse = **proc**(l: list[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string containing the *elements* of l unparsed by t$unparse in the form `[` *element*, ... `]`.
Modifies can have a secondary side-effect on l due to t$unparse.

2.8 Utility Type LONGINT

Objects of type `longint` are immutable and model mathematical integers. Only a subrange [`minlongint`, `maxlongint`] is represented. Objects of type `longint` use the integer literal form described in Section 2.5.

Operations

create = **proc**(i: sequence[byte]) **returns**(longint) **signals**(overflow)

Description Returns the signed integer corresponding to the unsigned integer value of i: $2^{0*bytesize} * s_k + 2^{1*bytesize} * s_{k-1} + \ldots$, where k is the last index of the sequence .
Signals overflow if the unsigned integer value of $i >$ `maxlongint`.
Sugared form of invocation is the integer literal.

min_value = **proc**() **returns**(longint)

Description Returns `minlongint`.

max_value = **proc**() **returns**(longint)

Description Returns `maxlongint`.

succ = **proc**(i: longint) **returns**(longint) **signals**(overflow)

Description Returns the successor of i.
Signals overflow if $i =$ `maxlongint`.

pred = **proc**(i: longint) **returns**(longint) **signals**(overflow)

Description Returns the predecessor of i.
Signals overflow if $i =$ `minlongint`.

add = **proc**(x, y: longint) **returns**(longint)

Description Integer addition.
Sugared form of invocation is x `+` y.

sub = **proc**(x, y: longint) **returns**(longint)

Description	Integer subtraction.
Sugared	form of invocation is x `-` y.

minus = **proc**(x: longint) **returns**(longint) **signals**(overflow)

Description	Returns the negative of x.
Signals	overflow if result would lie outside of the allowed range.
Sugared	form of invocation is `-` x.

mul = **proc**(x, y: longint) **returns**(longint)

Description	Integer multiplication.
Sugared	form of invocation is x `*` y.

div = **proc**(x, y: longint) **returns**(longint) **signals**(overflow, zero_divide)

Description	Returns the integer quotient of dividing x by y.
Signals	zero_divide if $y = 0$, overflow if $x =$ `minlongint` and $y = -1$.
Sugared	form of invocation is x `/` y.

mod = **proc**(x, y: longint) **returns**(longint) **signals**(zero_divide)

Description	Returns the remainder of dividing x by y.
Signals	zero_divide if $y = 0$.
Sugared	form of invocation is x `//` y.

abs = **proc**(x: longint) **returns**(longint) **signals**(overflow)

Description	Returns the absolute value of x.
Signals	overflow if x is `minlongint`

power = **proc**(x, y: longint) **returns**(longint) **signals**(negative_exponent)

Description	Returns x to the power y.
Signals	negative_exponent if $y < 0$.
Sugared	form of invocation is x `^` y

from_to_by = **iter**(f, t, b: longint) **yields**(longint)

Description	Yields the integers from f to t with increment b, that is, $f, f + b, ..., f + n * b$, where n is the largest integer such that $f + n * b \leq t$ if b is positive and $f + n * b \geq t$ otherwise. Terminates immediately if no such non-negative n exists. If $b = 0$, loops.
Sugared	form of invocation is `[`f .. t; b`]`

from_to = **iter**(f, t: longint) **yields**(longint)

Description Yields all integers from *f* to *t*.
Sugared form of invocation is [f .. t]

from_downto = **iter**(f, t: longint) **yields**(longint)

Description Yields all integers from *f* down to *t*.

lt = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if x is less than y.
Sugared form of invocation is x < y.

le = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if x is less than or equal to y.
Sugared form of invocation is x <= y.

ge = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if x is greater than or equal to y.
Sugared form of invocation is x >= y.

gt = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if x is greater than y.
Sugared form of invocation is x > y.

equal = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if $x = y$.
Sugared form of invocation is x = y.

similar = **proc**(x, y: longint) **returns**(bool)

Description Returns **true** if $x = y$.

copy = **proc**(x: longint) **returns**(longint)

Description Returns its argument.

from_int = **proc**(i: int) **returns**(longint)

Description Returns the corresponding longint.

to_int = **proc**(i: longint) **returns**(int) **signals**(overflow)

Description Returns the corresponding int.
Signals overflow if the result would lie outside of the allowed range of int.

parse = **proc**(s: string) **returns**(longint) **signals**(bad_format, overflow)

Description Converts string s representing an integer literal and returns the corresponding longint.
Signals overflow if the result would lie outside of the allowed range, bad_format if s is not a legal integer literal.

unparse = **proc**(x: longint) **returns**(string)

Description Produces a decimal integer literal corresponding to x.

2.9 Utility Type Generator MAYBE

For each type T there exists a type `maybe[T]`. Type `maybe[T]` is used to insert a bottom object to the parameter type T. If objects of the parameter type T are $\{t_1, t_2, \ldots\}$ then objects of type `maybe[T]` are $\{\perp, [t_1], [t_2], \ldots\}$. Objects of type `maybe[T]` are immutable.

Operations

create_undefined = **proc**(n: null) **returns**(maybe`[T]`)

Description Returns $\perp$.
Sugared form of invocation is ‘undefined.

create_value = **proc**(x: t) **returns**(maybe`[T]`)

Description Returns $[x]$.
Sugared form of invocation is ‘value(x).

defined = **proc**(m: maybe`[T]`) **returns**(bool)

Description Returns **true** if $m \neq \perp$.

get_value = **proc**(m: maybe`[T]`) **returns**(T) **signals**(undefined)

Description Returns the value of the object m.
Signals undefined if $m = \perp$.
Sugared form of invocation is m.value

equal = **proc**(m1, m2: maybe`[T]`) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if $m1$ and $m2$ have the same value (or both are the bottom object).
Sugared form of invocation is m1 `=` m2

similar = **proc**(m1, m2: maybe[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if $m1$ and $m2$ have similar values (or both are the bottom object).

copy = **proc**(m: maybe[T]) **returns**(maybe[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of m.

parse = **proc**(s: string) **returns**(maybe[T]) **signals**(syntax_error(string, int))
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a string that either has the form for T$parse or is "undefined".
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(m: maybe[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string produced by T$parse or "undefined" if m is the bottom object.

2.10 Utility Type Generator ORDSET

For each type T, there exists a type `ordset[T]`, if the type T has operations

gt, lt, equal, similar: **proctype**(T, T) **returns**(bool)

These operations should have no side-effects.

Objects of type `ordset[T]` are mutable and they model homogeneous sets of element the type T.

Operations

new = **proc**() **returns**(ordset[T])

Description Returns a new empty set.

create = **proc**(s: sequence[T]) **returns**(ordset[T])

Description Returns a new sequence containing the same elements as s.
Sugared form of invocation is the sequence constructor.

union = **proc**(a, b: ordset[T])

Description Inserts elements of b to a.
Modifies state of a.

intersection = **proc**(a, b: ordset[T])

Description Removes those elements from a that do not belong to b.
Modifies state of a.

difference = **proc**(a, b: ordset[T])

Description Removes elements of b from a.
Modifies state of a.

empty = **proc**(s: ordset[T]) **returns**(bool)

Description Returns **true** if s is empty.

size = **proc**(s: ordset[T]) **returns**(int)

Description Returns number of elements in s.

member = **proc**(s: ordset[T], e: T) **returns**(bool)

Description Returns **true** if e belongs to s.

insert = **proc**(s: ordset[T], e: T)

Description Inserts e to s.
Modifies state of s.

remove = **proc**(s: ordset[T], e: T) **signals**(empty)

Description Removes e from s.
Modifies state of s.
Signals empty if e is not a member of s.

least = **proc**(s: ordset[T]) **returns**(T) **signals**(empty)

Description Returns the smallest element in s.
Signals empty if there are no elements in s.

greatest = **proc**(s: ordset[T]) **returns**(T) **signals**(empty)

Description Returns the smallest element in s.
Signals empty if there are no elements in s.

elements = **proc**(s: ordset[T]) **returns**(T)

Description Yields all elements in s.

elements_down = **proc**(s: ordset[T]) **returns**(T)

Description Yields all elements in s (decreasing order).

from_to = **proc**(s: ordset[T], l, h: T) **returns**(T)

Description Yields in increasing order all elements e in s for whom $l \leq s \leq h$ holds.

from_downto = **proc**(s: ordset[T], h, l: T) **returns**(T)

Description Yields in decreasing order all elements e in s for whom $l \leq s \leq h$ holds.

equal = **proc**(a, b: ordset[T]) **returns**(bool)

Description Returns **true** if a and b are the same object.
Sugared form of invocation is a = b.

similar = **proc**(a, b: ordset[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if a and b have similar elements.
Modifies Can have a secondary side-effect on a or b due to T$similar.

equal_state = **proc**(a, b: ordset[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if a and b have equal elements.
Modifies Can have secondary side-effect on a or b due to T$equal.

subset = **proc**(a, b: ordset[T]) **returns**(bool)

Description Returns **true** if all elements of a belong to b.

copy = **proc**(s: ordset[T]) **returns**(ordset[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of s.
Modifies Can have a secondary side-effect on s due to T$copy.

copy_state = **proc**(s: ordset[T]) **returns**(ordset[T])

Description Returns a new set that has the member objects as s.

parse = **proc**(s: string) **returns**(ordset[T])
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a string of the form "[element, ...]" and returns the corresponding set.

unparse = **proc**(s: ordset[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string of the form " [element, ...] " representing s.

2.11 Utility Type Generator PARTIAL_RECORD

Objects of the types generated `partial_record` are quite like ordinary records, except that they can be contain uninitialized fields.

Operations

new = **proc**() **returns**(partial_record{F})

Description Returns a new record with all fields undefined.

create = **proc**(s: struct{F}) **returns**(partial_record{F})

Description Returns a new record r where $r_{N_i} = s_{N_i}$ for all names N_i.
Sugared form of invocation is the struct literal.

defined_N_i = **proc**(r: partial_record{F}) **returns**(bool)

Description Returns **true** if r_{N_i} is defined.

reset_N_i = **proc**(r: partial_record{F})

Description Sets r_{N_i} to be undefined.
Modifies state of r.

get_N_i = **proc**(r: partial_record{F}) **returns**(T_i) **signals**(undefined)

Description Returns r_{N_i}. There is a get_N_i operation for each field.
Signals undefined if the field has no value.
Sugared form of invocation is r.N_i .

set_N_i = **proc**(r: partial_record{F}, a: T_i)

Description Makes the state of r to differ from the old state in that the N_i-component is a. There is a set_N_i operation for each field.
Sugared form of invocation is r.N_i := a.

equal = **proc**(r1, r2: partial_record{F}) **returns**(bool)

Description Returns **true** if both arguments are the same object.
Sugared form of invocation is r1 = r2.

similar = **proc**(r1, r2: partial_record{F}) **returns**(bool)
where T_i **has** similar: **proctype**(T_i, T_i) **returns**(bool)

Description All components $r1_{N_i}$ and $r2_{N_i}$ are compared using T_i\$similar. If all comparisons return **true**, the result is **true**.

Modifies can have a secondary side-effect on $r1$ and $r2$ due to the T_i\$similars.

equal_state = **proc**(r1, r2: partial_record{F}) **returns**(bool)
where T_i **has** equal: **proctype**(T_i, T_i) **returns**(bool)

Description For each i components $r1_{N_i}$ and $r2_{N_i}$ are compared using T_i\$equal. If all comparisons return **true**, the result is **true**.

Modifies can have a secondary side-effect on $r1$ and $r2$ due to the T_i\$equals.

copy = **proc**(r: partial_record{F}) **returns**(partial_record{F})
where T_i **has** copy: **proctype**(T_i) **returns**(T_i)

Description Returns a new record $\langle T_1\$copy(r_{N_1}), ..., T_k\$copy(r_{N_k})\rangle$.

Modifies can have a secondary side-effect on r due to the T_i\$copy operations.

copy_state = **proc**(r: partial_record{F}) **returns**(partial_record{F})

Description Returns a new record with the same state as the argument.

from_record = **proc**(r: record{F}) **returns**(partial_record{F})

Description Return a partial record with the same field values.

to_record = **proc**(r: partial_record{F}) **returns**(record{F}) **signals**(undefined)

Description Return a partial record with the same field values.

Signals undefined if any of the fields is undefined.

parse = **proc**(s: string) **returns**(partial_record{F})
signals(syntax_error(string, int))
where T_i **has** parse: **proctype**(string) **returns**(T_i) **signals**(*)

Description Parses a list structure { *tag* : *element*, ... } where each *tag* is some N_i and the corresponding *element* is represented in the format of T_i\$parse. The field that are not present are set to be undefined. Returns a new record corresponding to s.

Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(r: partial_record{F}) **returns**(string)
where T_i **has** unparse: **proctype**(T_i) **returns**(string)

Description Returns a string { *tag* : *element*, ... } where *elements* are represented in form unparsed by T_i\$unparse. The fields that are undefined are left out.
Modifies can have a secondary side-effect on v due to t\$unparse.

2.12 Utility Type Generator PRIO_QUEUE

For each type T that has operation

better: **proctype**(T, T) **returns**(bool)

there exists a type `prio_queue[T]`. Objects of type `prio_queue[T]` model priority queues of homogeneous object and they are mutable. Operation `better` of type T should have no side-effects.

new = **proc**() **returns**(prio_queue[T])

Description Returns a new empty priority queue.

create = **proc**(s: sequence[T]) **returns**(prio_queue[T])

Description Returns a new priority queue that contains the elements of s.
Sugared form of invocation is the sequence constructor.

empty = **proc**(q: prio_queue[T]) **returns**(bool)

Description Returns **true** if q is empty.

size = **proc**(q: prio_queue[T]) **returns**(int)

Description Returns number of elements in q.

peek = **proc**(q: prio_queue[T]) **returns**(T) **signals**(empty)

Description Returns the best element in q.
Signals empty if there is no elements in q.

put = **proc**(q: prio_queue[T], e: t)

Description Inserts element e to queue q.
Modifies state of q.

get = **proc**(q: prio_queue[T]) **signals**(empty)

Description Removes and returns the best element in *q*.
Modifies state of *q*.
Signals empty if there is no elements in *q*.

equal = **proc**(a, b: prio_queue[T]) **returns**(bool)

Description Returns **true** if *a* and *b* are the same object.
Sugared form of invocation is a = b.

similar = **proc**(a, b: prio_queue[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if *a* and *b* contain similar objects.
Modifies Can have a secondary side-effect on *a* or *b* due to T$similar.

equal_state = **proc**(a, b: prio_queue[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if *a* and *b* contain the same objects.
Modifies Can have a secondary side-effect on *a* or *b* due to T$equal.

copy = **proc**(q: prio_queue[T]) **returns**(prio_queue[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of *q*.
Modifies Can have secondary side-effect on *q* due to T$copy.

copy_state = **proc**(q: prio_queue[T]) **returns**(prio_queue[T])

Description Returns a priority queue that has the same contents as *q*.

parse = **proc**(s: string) **returns**(prio_queue[T])
signals(syntax_error(string, int))
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a string of the form "[element, ...]" and returns the corresponding queue.
Signals syntax_error(*m*, *p*) if the form of *s* is invalid (*m* is the error message and *p* the position of the error in *s*).

unparse = **proc**(q: prio_queue[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string of the form "[element, ...]" representing elements of *q* in best first order.

2.13 Utility Type Generator SET

For each type T that has operation

equal: **proctype**(T, T) **returns**(bool)

there exists a type `set[T]`. Objects of a type `set[T]` are mutable and they model homogeneous sets of elements of the type T. Operation `equal` of type T should have no side-effects.

Operations

new = **proc**() **returns**(set[T])

> **Description** Returns a new empty set.

create = **proc**(s: sequence[T]) **returns**(set[T])

> **Description** Returns a new set containing the same elements as *s*.
> **Sugared** form of invocation is the sequence constructor.

union = **proc**(a, b: set[T]) **returns**(set[T])

> **Description** Returns a set that contain all elements that belong to *a* or *b*.

intersection = **proc**(a, b: set[T]) **returns**(set[T])

> **Description** Returns a set that contain all elements that belong to both *a* and *b*.

difference = **proc**(a, b: set[T]) **returns**(set[T])

> **Description** Returns a set that contain all elements that belong to *a*, but not to *b*.

empty = **proc**(s: set[T]) **returns**(bool)

> **Description** Returns **true** if *s* is empty.

size = **proc**(s: set[T]) **returns**(int)

> **Description** Returns number of elements in *s*.

member = **proc**(s: set[T], e: T) **returns**(bool)

> **Description** Returns **true** if *e* belongs to *s*.

insert = **proc**(s: set[T], e: T)

Description Inserts *e* to *s*.
Modifies state of *s*.

remove = **proc**(s: set[T], e: T) **signals**(not_found)

Description Removes *e* from *s*.
Modifies state of *s*.
Signals not_found if *e* is not a member of *s*.

get = **proc**(s: set[T]) **returns**(T) **signals**(empty)

Description Returns an arbitrary element of *s*. (Note: this is *not* a random function.)
Signals empty if there are no elements in *s*.

elements = **proc**(s: set[T]) **returns**(T)

Description Yields all elements in *s*.

equal = **proc**(a, b: set[T]) **returns**(bool)

Description Returns **true** if *a* and *b* are the same object.
Sugared form of invocation is a = b

similar = **proc**(a, b: set[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description Returns **true** if *a* and *b* have similar elements.
Modifies Can have a secondary side-effect on *a* or *b* due to T$similar.

equal_state = **proc**(a, b: set[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description Returns **true** if *a* and *b* have equal elements.
Modifies Can have a secondary side-effect on *a* or *b* due to T$equal.

subset = **proc**(a, b: set[T]) **returns**(bool)

Description Returns **true** if all elements of *a* belong to *b*.

copy = **proc**(s: set[T]) **returns**(set[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of *s*.
Modifies Can have a secondary side-effect on *a* due to T$copy.

copy_state = **proc**(s: set[T]) **returns**(set[T])

Description Returns a new set that has the same the members objects as s.

parse = **proc**(s: string) **returns**(set[T])
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a string of the form "[element, ...]" and returns the corresponding set.

unparse = **proc**(s: set[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns a string of the form "[element, ...]" representing s.

2.14 Utility Type Generator STACK

For each type T there exists a type `stack[T]`. Objects of type `stack[T]` are mutable. The index of the topmost element is 0.

Operations

new = **proc**() **returns**(stack[T])

Description Returns a new empty stack.

create = **proc**(s: sequence[T]) **returns**(stack[T])

Description Returns a stack containing the elements of s, the last one on the top.
Sugared form of invocation is the sequence constructor.

size = **proc**(s: stack[T]) **returns**(int)

Description Returns count of elements stored in the stack.

empty = **proc**(s: stack[T]) **returns**(bool)

Description Returns **true** if $size_s = 0$.

push = **proc**(s: stack[T], item: t)

Description Adds a new element on the top of the stack.
Modifies State of s.

pop = **proc**(s: stack[T]) **returns**(t) **signals**(empty)

Description	Removes the top element of the stack.
Modifies	status of s.
Signals	empty when size(stack) = 0.

get_top = **proc**(s: stack[T]) **returns**(t) **signals**(empty)

Description	Returns the top element of the stack.
Signals	empty when size(stack) = 0.
Sugared	form of invocation is s.top

set_top = **proc**(s: stack[T], e: t) **signals**(empty)

Description	Replaces the top element of the stack.
Signals	empty when size(stack) = 0.
Sugared	form of invocation is s.top := e

fetch = **proc**(s: stack[T], i: int) **returns**(t) **signals**(bounds)

Description	Returns the i^{th} element of the stack.
Signals	bounds if $i \notin 0..size_s - 1$
Sugared	form of invocation is s[i].

elements = **iter**(s: stack[T]) **yields**(T)

Description	Yields all elements from top to bottom.

indexes = **iter**(s: stack[T]) **yields**(int)

Description	Yields all indexes starting from 0.

equal = **proc**(s1, s2: stack[T]) **returns**(bool)

Description	Returns **true** if $s1$ and $s2$ are the same stack.
Sugared	form of invocation is s1 = s2

similar = **proc**(s1, s2: stack[T]) **returns**(bool)
where T **has** similar: **proctype**(T, T) **returns**(bool)

Description	Returns **true** if stacks are similar.
Modifies	Can have a secondary side-effect on $s1$ or $s2$ due to T$similar.

equal_state = **proc**(s1, s2: stack[T]) **returns**(bool)
where T **has** equal: **proctype**(T, T) **returns**(bool)

Description	Returns **true** if stacks have equal states.
Modifies	Can have a secondary side-effect on $s1$ or $s2$ due to T$similar.

copy = **proc**(s: stack[T]) **returns**(stack[T])
where T **has** copy: **proctype**(T) **returns**(T)

Description Returns a copy of s.
Modifies can have a secondary side-effect on s due to T$copy.

copy_state = **proc**(s: stack[T]) **returns**(stack[T])

Description Returns a new stack that has the same state as s.

to_sequence = **proc**(s: stack[T]) **returns**(sequence[t])

Description Returns s as a sequence (topmost element last).

parse = **proc**(s: string) **returns**(stack[T]) **signals**(syntax_error(string, int))
where T **has** parse: **proctype**(string) **returns**(T) **signals**(*)

Description Parses a string of the form `[` *element*, ... `]`, where the topmost element comes first.
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(s: stack[T]) **returns**(string)
where T **has** unparse: **proctype**(T) **returns**(string)

Description Returns string of the form `[` *element*, ... `]`, where the top element comes first.
Modifies Can have a secondary side-effect on s due to t$unparse.

2.15 Utility Type Generator TABLE

For each pair types K and E there exists a type `table[K, E]`, if the type K has operations:

hash: **proctype**(K) **returns**(int)
equal: **proctype**(K, K) **returns**(bool)

Objects of type `table[K, E]` are mutable and they model associative memories (hash tables).

Operations

new = **proc**() **returns**(table[K, E])

Description Returns a new empty table.

empty = **proc**(t: table[K, E]) **returns**(bool)

Description Returns **true** if table is empty.

size = **proc**(t: table[K, E]) **returns**(int)

Description Returns number of pairs in t.

fetch = **proc**(t: table[K, E], i: K) **returns**(E) **signals**(not_found)

Description Returns the element associated with key i.
Signals not_found if there is no element associated with i.
Sugared form of invocation is t[i]

store = **proc**(t: table[K, E], i: K, v: E)

Description Associates element v with key i.
Modifies state of t.
Sugared form of invocation is t[i] := v

remove = **proc**(t: table[K, E], i: K) **returns**(E) **signals**(not_found)

Description Removes and returns element associated with i.
Modifies state of t.
Signals not_found if there is no element associated to i.

legal_key = **proc**(t: table[K, E], i: K) **returns**(bool)

Description Returns **true** if there is an element associate with key i.

flush = **proc**(t: table[K, E])

Description Removes all key–element pairs from the table.
Modifies state of t.

cond_flush = **proc**(t: table[K, E], p: **proctype**(K) returns(bool))

Description Removes all key–element, for whom p(key) is **true**.
Modifies state of t.

elements = **iter**(t: table[K, E]) **yields**(E)

Description Yields all elements in t.

indexes = **iter**(t: table[K, E]) **yields**(K)

Description Yields all keys in t.

pairs = **iter**(t: table[K, E]) **yields**(K, E)

Description Yields all key–element pairs in t.

equal = **proc**(a, b: table[K, E]) **returns**(bool)

Description Returns **true** if a and b are the same object.
Sugared form of invocation is a = b

similar = **proc**(a, b: table[K, E]) **returns**(bool)
where E **has** similar: **proctype**(E, E) **returns**(bool)

Description Returns **true** if a and b have similar key–element pairs.
Modifies can have a secondary side-effect on a or b due to K$similar and E$similar.

equal_state = **proc**(a, b: table[K, E]) **returns**(bool)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Returns **true** if a and b have equal key–element pairs.
Modifies can have a secondary side-effect on a or b due to K$equal and E$equal.

copy = **proc**(t: table[K, E]) **returns**(table[K, E])
where E **has** copy: **proctype**(E) **returns**(E)

Description Returns a copy of t.
Modifies can have a secondary side-effect on t due to K$copy and E$copy.

copy_state = **proc**(t: table[K, E]) **returns**(table[K, E])

Description Returns a new table that has the same key–element pairs as t.

parse = **proc**(s: string) **returns**(table[K, E]) **signals**(syntax_error(string, int))
where E **has** parse: **proctype**(string) **returns**(E) **signals**(*)

Description Parses a string of the form "[key: element, ...]" and returns the corresponding table.
Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(t: table[K, E]) **returns**(string)
where E **has** unparse: **proctype**(E) **returns**(string)

Description Returns a string of the form "[key: element, ...]" representing table t.
Modifies can have a secondary side-effect on t due to K$unparse and E$unparse.

2.16 Utility Type TIME

Objects of type `time` are immutable and represent amounts of elapsed time. A time object consists of six components that are always canonized in the ranges: `days [0..]`, `hours [0..23]`, `minutes [0..59]`, `seconds [0..59]`, `millis [0..999]`, and `micros [0..999]`. Note that elapsed time cannot be negative.

create = **proc**(t: struct{days, hours, minutes, seconds, millis, micros: int })
returns(time) **signals**(negative_time)

Description	Returns the time represented by *t*.
Signals	negative_time if a negative time value is represented by the struct.
Sugared	form of invocation is the struct literal.

add = **proc**(t1, t2: time) **returns**(time)

Description	Addition.
Sugared	form of invocation is *t1* + *t2*

sub = **proc**(t1, t2: time) **returns**(time) **signals**(negative_time)

Description	Subtraction.
Signals	negative_time if the result would be negative.
Sugared	form of invocation is *t1* - *t2*

mul = **proc**(t: time, n: int) **returns**(time) **signals**(negative_time)

Description	Returns the time multiplied by *n*.
Signals	negative_time if the result would be negative.

get_days = **proc**(t: time)

Description	Returns number of days.
Sugared	form of invocation is t.days

get_hours = **proc**(t: time)

Description	Returns number of hours.
Sugared	form of invocation is t.hours

get_minutes = **proc**(t: time)

Description	Returns number of minutes.
Sugared	form of invocation is t.minutes

get_seconds = **proc**(t: time)

Description	Returns number of seconds.
Sugared	form of invocation is t.seconds

get_millis = **proc**(t: time)

Description Returns number of millis.
Sugared form of invocation is t.millis

get_micros = **proc**(t: time)

Description Returns number of micros.
Sugared form of invocation is t.micros

lt = **proc**(t1, t2: time)

Description Returns **true** if t1 is less than t2.
Sugared form of invocation is *t1* <*t2*

le = **proc**(t1, t2: time)

Description Returns **true** if t1 is less than or equal to t2.
Sugared form of invocation is *t1* >= *t2*

ge = **proc**(t1, t2: time)

Description Returns **true** if t1 is greater than or equal to t2.
Sugared form of invocation is *t1* >= *t2*

gt = **proc**(t1, t2: time)

Description Returns **true** if t1 is greater than t2.
Sugared form of invocation is *t1* >*t2*

equal = **proc**(t1, t2: time)

Description Return **true** if $t1 = t2$.
Sugared form of invocation is *t1* = *t2*

similar = **proc**(t1, t2: time)

Description Return **true** if $t1 = t2$.

copy = **proc**(t1, t2: time)

Description Returns its argument.

to_struct = **proc**(t: time)
returns(struct{days, hours, minutes, seconds, millis, micros: int })

Description Returns time represented by a struct.

parse = **proc**(s: string) **returns**(time) **signals**(bad_format)

Description Parses a time of the form "days:hours:mins:secs.fraction" and returns the result. Any of the leading fields and the fraction part can be left out, e.g. "123" means 2 minutes 3 seconds.

Signals bad_format if s does not represent a legal time value.

unparse = **proc**(t: time) **returns**(string)

Description Returns the canonized time represented by a string of form "days:hours:mins:secs.fraction". Leading zeros and trailing fraction zeros are stripped of.

2.17 Utility Type Generator TRIE

For each type I and type L there exists a type `trie[L, I]`, if the type L has operations:

gt, lt, equal, similar: **proctype**(L, L) **returns**(bool)

These operations should have no side-effects.

Trie is a tree indexed by paths of labels from the top, not by single labels. The abstraction of a trie is very much alike a partial function from the set of paths to the set of items. The items are of the domain type I and the paths of the range type `sequence[L]`. A trie is thus a set of pairs $(\{\ l_1, l_2, \ldots\}, i)$, where $l_n \in$ L and $i \in$ I. Objects of trie types are mutable.

Operations

new = **proc**() **returns**(trie[L, I])

Description Returns a new empty trie.

empty = **proc**(t: trie[L, I]) **returns**(bool)

Description Returns **true** if size(t) = 0.

size = **proc**(t: trie[L, I]) **returns**(int)

Description Returns the number of items stored in trie t.

count = **proc**(t: trie[L, I]) **returns**(int)

Description Returns the number of subtries.

legal_path = **proc**(t: trie[L, I], p: sequence[L]) **returns**(bool)

Description Returns **true** if an item with path p exists.

legal_label = **proc**(t: trie[L, I], n: L) **returns**(bool)

Description Returns **true** if a subtrie with label n exists.

fetch = **proc**(t: trie[L, I], p: sequence[L]) **returns**(I) **signals**(not_found)

Description Returns the item that is stored to path p in t.
Signals not_found if path p does not exist or no item is stored there.
Sugared form of invocation is t[p]

get = **proc**(t: trie[L, I], n: L) **returns**(trie[L, I]) **signals**(not_found)

Description Returns the subtrie at label n.
Signals not_found if label n does not exist.

store = **proc**(t: trie[L, I], p: sequence[L], x: I)

Description Stores the item x to path p in t. Note that if an item exists at p, it is replaced with x.
Modifies state of t.
Sugared form of invocation is t[p] := x

set = **proc**(t: trie[L, I], p: sequence[L], s: trie[L, I])

Description Stores trie s (as a subtrie) to end of path p in t. Note that if a subtrie exists at p, it is replaced with s.
Modifies state of t.

remove = **proc**(t: trie[L, I], p: sequence[L]) **signals**(not_found)

Description Removes the item that is stored to path p in t.
Modifies state of t.
Signals not_found if path p does not exist or no item is stored there.

delete = **proc**(t: trie[L, I], n: L) **returns**(trie[L, I]) **signals**(not_found)

Description Removes the subtrie with label n.
Modifies state of t.
Signals not_found if label n does not exist.

paths = **iter**(t: trie[L, I]) **yields**(sequence[L])

Description Yields all paths defined in t in lexicographic order.

labels = **iter**(t: trie[L, I]) **yields**(L)

Description Yields all labels from the top node.

items = **iter**(t: trie[L, I]) **yields**(I)

Description Yields all items stored in *t* in alphabetical order of their paths.

subtries = **iter**(t: trie[L, I]) **yields**(trie[L, I])

Description Yields all subtries of the top node.

item_pairs = **iter**(t: trie[L, I]) **yields**(sequence[L], I)

Description Yields all items stored in *t* and their paths in alphabetical order of the paths. Note that there can be paths in the trie where no items are stored.

subtrie_pairs = **iter**(t: trie[L, I]) **yields**(L, trie[L, I])

Description Yields all labels and subtries of *t*.

equal = **proc**(t1, t2: trie[L, I]) **returns**(bool)

Description Returns **true** if *t*1 and *t*2 are the same object.
Sugared form of invocation is t1 = t2

similar = **proc**(t1, t2: trie[L, I]) **returns**(bool)
where I **has** similar: **proctype**(I, I) **returns**(bool)

Description Returns **true** if *t*1 and *t*2 have similar items at similar paths.

equal_state = **proc**(t1, t2: trie[L, I]) **returns**(bool)
where I **has** equal: **proctype**(I, I) **returns**(bool)

Description Returns **true** if *t*1 and *t*2 have the same state (i.e., have equal items at equal paths).

copy = **proc**(t: trie[L, I]) **returns**(trie[L, I])
where I **has** copy: **proctype**(I) **returns**(I)

Description Returns a copy of *t*.

copy_state = **proc**(t: trie[L, I]) **returns**(trie[L, I])

Description Returns a trie that has the same state as *t* (i.e. it has the equal items at equal paths).

parse = **proc**(s: string) **returns**(trie[L, I]) **signals**(syntax_error(string, int))
where I **has** parse: **proctype**(string) **returns**(I) **signals**(*)

Description Parses a string of the form `[ [` label, ... `]:` item1, ... `]` and returns the corresponding trie.

Signals syntax_error(m, p) if the form of s is invalid (m is the error message and p the position of the error in s).

unparse = **proc**(t: trie[L, I]) **returns**(string)
where I **has** unparse: **proctype**(I) **returns**(string)

Description Returns a string representing trie t.

Modifies can have a secondary side-effect on t due to I$unparse and L$unparse.

3. Utility Routines

binary_search = **proc**[T, I, E: type](tuple: T, low, high: I, element: E)
returns(I) **signals**(not_found, not_possible(string))
where T **has** fetch: **proctype**(T, I) **returns**(E) **signals**(*)
where E **has** gt: **proctype**(E, E) **returns**(bool)
where I **has** gt: **proctype**(I, I) **returns**(bool),
add, sub, div: **proctype**(I, I) **returns**(I) **signals**(*),
create: **proctype**(sequence[byte]) **returns**(I) **signals**(*)

Description Searches from `tuple` the element x that is as great as *element*. Search is done between indexes *low* and *high* (inclusive). The elements in the tuple must be ordered in increasing order of indexes.

Signals not_found when there exists no such element in the given interval, not_possible(w) when any of the required operations signals (w contains the signal name).

elem_count = **proc**[T, E: type](tuple: T, element: E) **returns**(int)
where T **has** elements: **itertype**(T) **yields**(E)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Returns number of occurences of *element* in *tuple*.

first_index = **proc**[T, I, E: type](element: E, tuple: T) **returns**(index)
signals(not_found, not_possible(string))
where T **has** indexes: **itertype**(T) **yields**(I),
fetch: **proctype**(T, I) **returns**(E) **signals**(*)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Finds the first occurence of *element* in *tuple*.

Signals not_found when there is no such element, not_possible(w) if t$fetch signals ($w$ is the name of the signal).

first_pos = **proc**[T, I, E: type](tuple1, tuple2: T) **returns**(index)
signals(not_found, not_possible(string))
where T **has** indexes: **itertype**(T) **yields**(I),
fetch: **proctype**(T, I) **returns**(E) **signals**(*)
where I **has** add, sub: **proctype**(I, I) **returns**(I) **signals**(*)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Finds the first occurence of *tuple1* in *tuple2*.

Signals not_found when there is no such subtuple, not_possible(w) if any of the required routines signals (w is the name of the signal).

heap_sort = **proc** [T, I, E: type] (tuple: T) **signals**(not_possible(string))
where T **has** store: **proctype**(T, I, E) **signals**($*$),
elements: **itertype**(T) **yields**(E),
indexes: **itertype**(T) **yields**(I),
where E **has** lt: **proctype**(E, E) **returns**(bool)

Description Sorts given tuple to increasing order as determined by element$lt.
Modifies state of *t*.
Signals not_possible(*what*) if any of the required routines signals (*what* is the name of the signal).

last_index = **proc** [T, I, E: type] (element: E, tuple: T) **returns**(index)
signals(not_found, not_possible(string))
where T **has** indexes_down: **itertype**(T) **yields**(I),
fetch: **proctype**(T, I) **returns**(E) **signals**($*$)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Finds the last occurence of *element* in *tuple*.
Signals not_found when there is no such element, not_possible(*w*) if t$fetch signals (*w* is the name of the signal).

last_pos = **proc** [T, I, E: type] (tuple1, tuple2: T) **returns**(index)
signals(not_found, not_possible(string))
where T **has** indexes_down: **itertype**(T) **yields**(I),
fetch: **proctype**(T, I) **returns**(E) **signals**($*$)
where I **has** sub: **proctype**(I, I) **returns**(I) **signals**($*$)
where E **has** equal: **proctype**(E, E) **returns**(bool)

Description Finds the last occurence of *tuple1* in *tuple2*.
Signals not_found when there is no such subtuple, not_possible(*w*) if any of the required routines signals (*w* is the name of the signal).

lex_scan = **proc**(l, s: string) **returns**(sequence [string])

Description Splits *l* to substring that do not contain characters in *s* (i.e. *s* contains the separators).

permutations = **iter** [T: type] (a: list [T]) **yields**(list [T] , bool)

Description Yields all permutations of elements in *a* and the parity of each permutation (**true** = even).

sort = **proc**[T, I, E: type](tuple: T, low, high: I)
signals(not_possible(string))
where T **has** fetch: **proctype**(T, I) **returns**(E) **signals**(*),
store: **proctype**(T, I, E) **signals**(*)
where I **has** from_to: **itertype**(I, I) **yields**(I),
where E **has** gt: **proctype**(E, E) **returns** (bool)

Description Sorts subtuple $a_{low}, ..., a_{high}$ to increasing order of elements (as determined by element$gt).
Modifies state of a.
Signals not_possible(w) if any of the required routines signals (w contains the name of the signal).

wildcard_match = **proc**(p, s: string) **returns**(bool) **signals**(empty_pattern)

Description Returns **true** if string s matches wild card pattern p. The allowed wild cards are * and %, * matches any string of any length, % matches any single character. Wild cards cannot be quoted.
Signals empty_pattern if $p =$ "".

4. Extensions to Data Types

stream_ask = **proc**(prompt: string) **returns**(string)
signals(end_of_file, not_possible(string))

Description	Writes a prompt to primary output and reads the answer.
Signals	end_of_file, not_possible(string) reading or writing is not possible.

stream_open = **proc**(prompt: string, mode: stream_mode) **returns**(stream)
signals(not_possible(string))

Description	Writes *prompt*, reads a file name, and tries to open it. Returns the opened stream.
Signals	not_possible(*why*) if an error occurs.

string_capitalize = **proc**(s: string) **returns**(string)

Description Returns *s* capitalized.

string_compare = **proc**(a, b: string) **returns**(int, int, int)

Description Compares two strings. Returns start index and length of different substrings. If strings are similar zeros are returned. Return order start, length of substring in *a*, length of substring in *b*.

string_hash = **proc**(s: string) **returns**(int)

Description Returns a integer hashing key for string *s*.

string_strip = **proc**(s: string) **returns**(string)

Description Strips leading and trailing spaces and tabs.

string_strip_low = **proc**(s: string) **returns**(string)

Description Strips leading spaces and tabs.

string_strip_high = **proc**(s: string) **returns**(string)

Description Strips trailing spaces and tabs.

string_to_lower = **proc**(s: string) **returns**(string)

Description Returns *s* in lower case.

string_to_upper = **proc**(s: string) **returns**(string)

Description Returns s in upper case.

string_unify_spacing = **proc**(s: string) **returns**(string)

Description Unifies spacing in a string, i.e. replaces sequences of blanks and tab with a single blank.