Appendix A: Built-in Types

A.1 Introduction

XE has some types and type generators built-in. Built-in means that these types have some special properties (see Section 1.4) and the compiler is to some extent aware of their existence and implementation. Some built-in types are implemented in a special way to make them more efficient.

The built-in types are described in this appendix. 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**. All operations of built-in types are *total* – they do not have any requirements on their arguments to work properly. If some operation can not be executed, for example due to lack of memory space, **failure** is signaled.

Some operations have side effects, i.e., modify their argument objects. These operations have item **modifies** that lists the arguments modified. Not all of the operations with item **modifies** always modify their arguments, because some side effects are secondary. Secondary side effect means that the operations in the **where** clause can modify the arguments, but the operation itself does not.

If an operation can be invoked using the syntactic sugar of XE then the sugared form of invocation is given in item **sugared**.

All built-in types have the `dump` and `undump` operations as described in Section 6.3.

A.2 Built-in Type NULL

Type `null` has only one object `nil`. It is used as a place holder for types like those generated by `oneof` and `variant` to create void fields. Usage of `null` enables termination of recursive data structures (e.g., trees).

Operations

create_nil = **proc**() **returns**(null)
signals()

Description Returns the `nil` object.

equal = **proc**(n1, n2: null)

Description Returns always **true**.
Sugared form of invocation is n1 `=` n2.

similar = **proc**(n1, n2: null) **returns**(bool)

Description Returns always **true**.

copy = **proc**(n: null) **returns**(null)

Description Returns the `nil` object.

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

Description Returns the `nil` object.
Signals bad_format if the lower case form of *s* is not "nil".

unparse = **proc**(n: null) **returns**(string)

Description Returns the string "nil"

A.3 Built-in Type BOOL

The two immutable objects of type `bool`, with literals **true** and **false**, represent logical truth values.

Type `bool` is inherently bound into the language. For operating on boolean expressions there exist two exceptional operations **cand** and **cor**, that are not operations of type `bool`, but constructs of the language. They return respectively the logical `and` and `or` of their left and right sides, but they evaluate the right side only if the result is not deduced by evaluating the left side. (The **c** in their names stands for conditional evaluation.)

Operations

bitand = **proc**(b1, b2: bool) **returns**(bool)

Description Returns the logical and of *b*1 and *b*2.
Sugared form of invocation is b1 `&` b2.

bitor = **proc**(b1, b2: bool) **returns**(bool)

Description	Returns the logical `or` of $b1$ and $b2$.
Sugared	form of invocation is b1 | b2.

bitnot = **proc**(b: bool) **returns**(bool)

Description	Returns the logical negation of b.
Sugared	form of invocation is ~ b.

equal = **proc**(b1, b2: bool) **returns**(bool)

Description	Returns **true** if $b1 = b2$.
Sugared	form of invocation is b1 = b2.

similar = **proc**(b1, b2: bool) **returns**(bool)

Description	Returns **true** if $b1 = b2$.

copy = **proc**(b: bool) **returns**(bool)

Description	Returns its argument.

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

Description	Returns the value of the boolean literal s.
Signals	bad_format when the lower case form of s is not "true" or "false".

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

Description	Produces the lower case boolean literal ("true" or "false") corresponding to b.

A.4 Built-in Type BYTE

Objects of type `byte` are immutable and they model bit vectors of `BYTESIZE` bits. Typically `BYTESIZE` is 8. They can also be treated as a subrange of positive integers (i.e. unsigned integers). The exact range is from 0 to $2^{\texttt{BYTESIZE}} - 1$. The addition and subtraction of bytes follow the underlying machine implementation. Thus overflow in these operations is not signaled. Instead (in two's complement machines) these operations are typically performed modulo 256.

Bytes use the integer literal form described in Section 2.5.

Operations

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

> **Description** Returns the byte given in the sequence i. If i is empty then returns 0.
> **Signals** overflow if i has more than one byte.
> **Sugared** form of invocation is the integer literal.

min_value = **proc**() **returns**(byte)

> **Description** Returns the byte where all bits are zeros.

max_value = **proc**() **returns**(byte)

> **Description** Returns the byte where all bits are ones.

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

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

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

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

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

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

div = **proc**(x, y: byte) **returns**(byte) **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: byte) **returns**(byte) **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.

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

Description Bitwise and-operation of x and y.
Sugared form of invocation is x & y.

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

Description Bitwise or-operation of x and y.
Sugared form of invocation is x | y.

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

Description Bitwise exclusive or-operation of x and y.

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

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

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

Description Returns the byte where the bits of x are shifted left n positions (the most significant bit is the left most). The n leftmost bits are shifted ”out”.

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

Description Returns the byte where the bits of x are shifted right n positions (the most significant bit is the left most). The n rightmost bits are shifted ”out”.

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

Description Returns the byte where the bits of x are rotated left n positions (the most significant bit is the left most).

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

Description Returns the byte where the bits of x are rotated right n positions (the most significant bit is the left most).

fetch = **proc**(x: byte, 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 $i \notin \{0, ..., \texttt{BYTESIZE} -1\}$.
Sugared form of invocation is `x[i]`.

replace = **proc**(x: byte, i: int, b: bool) **returns**(byte) **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 $i \notin \{0, ..., \texttt{BYTESIZE} -1\}$.

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

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

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

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

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

Description Yields all bytes from f down to t.

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

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

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

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

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

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

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

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

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

> **Description** Returns its argument.

to_char = **proc**(b: byte) **returns**(char)

> **Description** Returns the character corresponding to b.

to_int = **proc**(x: byte) **returns**(int)

> **Description** Returns the int equal to x.

to_word = **proc**(x: byte) **returns**(word)

> **Description** Returns the word having the same value as x.

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

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

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

> **Description** Produces a decimal integer literal corresponding to b.

A.5 Built-in Type INT

Objects of type `int` are immutable and model mathematical integers. Only a subrange [`MININT`, `MAXINT`] is represented. Addition and subtraction follow the implementation of the underlying machine instruction set. Thus normally an overflow exception is not generated but on a two's complement machine addition and subtraction can be considered to be performed modulo the range of integers.

Objects of type `int` use the integer literal form described in Section 2.5.

Operations

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

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

min_value = **proc**() **returns**(int)

Description	Returns `MININT`.

max_value = **proc**() **returns**(int)

Description	Returns `MAXINT`.

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

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

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

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

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

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

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

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

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

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

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

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

mod = **proc**(x, y: int) **returns**(int) **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: int) **returns**(int) **signals**(overflow)

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

power = **proc**(x, y: int) **returns**(int) **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: int) **yields**(int)

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: int) **yields**(int)

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

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

Description Yields all integers from f down to t.

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

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

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

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

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

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

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

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

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

Description Returns its argument.

to_byte = **proc**(x: int) **returns**(byte) **signals**(overflow)

Description Returns the byte equal to x.
Signals overflow if $x \notin \{0, ..., 2^{bytesize} - 1\}$.

to_char = **proc**(i: int) **returns**(char) **signals**(illegal_char)

Description Returns the character corresponding to i.
Signals illegal_char if the result would be outside of the allowed range of characters.

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

Description Returns the word having the same value as x.
Signals overflow if x is negative.

make_word = **proc**(x: int) **returns**(word)

Description Returns the word having the same bit pattern as x.

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

Description Converts string s representing an integer literal and returns the corresponding integer.
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: int) **returns**(string)

Description Produces a decimal integer literal corresponding to x.

A.6 Built-in Type WORD

Objects of type `word` are immutable and they model bit vectors of `WORDSIZE` bits. Typically `WORDSIZE` is 16 or 32. They can also be treated as a subrange of positive integers (i.e. unsigned integers). The exact range is from 0 to $2^{\texttt{WORDSIZE}} - 1$. The addition and subtraction of words follow the underlying machine implementation. Thus overflow in these operations is not signaled. Instead (in two's complement machines) these operations are typically performed modulo $2^{\texttt{WORDSIZE}}$.

Words use the integer literal form described in Section 2.5.

Operations

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

Description	Returns the word given in the sequence i, most significant byte at position 0.
Signals	overflow if bytes contained in i together hold more than `WORDSIZE` bits.
Sugared	form of invocation is the integer literal.

min_value = **proc**() **returns**(word)

Description	Returns the word where all bits are zeros.

max_value = **proc**() **returns**(word)

Description	Returns the word where all bits are ones.

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

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

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

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

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

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

div = **proc**(x, y: word) **returns**(word) **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: word) **returns**(word) **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.

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

> **Description** Bitwise and-operation of x and y.
> **Sugared** form of invocation is x & y.

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

> **Description** Bitwise or-operation of x and y.
> **Sugared** form of invocation is x | y.

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

> **Description** Bitwise exclusive or-operation of x and y.

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

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

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

> **Description** Returns the word where the bits of x are shifted left n positions (the most significant bit is the left most). The n leftmost bits are shifted "out".

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

> **Description** Returns the word where the bits of x are shifted right n positions (the most significant bit is the left most). The n rightmost bits are shifted "out".

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

> **Description** Returns the word where the bits of x are rotated left n positions (the most significant bit is the left most).

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

> **Description** Returns the word where the bits of x are rotated right n positions (the most significant bit is the left most).

fetch = **proc**(x: word, 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 $i \notin \{0, ..., \texttt{WORDSIZE} -1\}$.
Sugared form of invocation is `x[i]`.

replace = **proc**(x: word, i: int, b: bool) **returns**(word) **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 $i \notin \{0, ..., \texttt{WORDSIZE} -1\}$.

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

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

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

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

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

Description Yields all words from f down to t.

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

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

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

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

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

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

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

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

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

Description Returns its argument.

to_byte = **proc**(x: word) **returns**(byte) **signals**(overflow)

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

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

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

make_int = **proc**(x: word) **returns**(int)

Description Returns the integer having the same bit pattern as x.

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

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

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

Description Produces a decimal integer literal corresponding to b.

A.7 Built-in Type CHAR

Objects of type `char` are immutable, and represent characters. Characters are numbered, but no standard numbering is guaranteed (typically it is ASCII). However, the following properties are guaranteed:

- 'a' < 'b' < ... < 'z'
- 'A' < 'B' < ... < 'Z'
- '0' < '1' < ... < '9'

Letters a to z and A to Z have indexes 0..25 in their alphabetical order (for example the index of 'b' and 'B' is 1). Digits have indexes 0..9 according to their values. Objects of type `char` use the character literal form '*c*' described in Section 2.5.

Operations

create = **proc**(c: char) **returns**(char)

> **Description** Returns its argument (useful in **where**-clauses).
> **Sugared** form of invocation is the character literal.

min_value = **proc**() **returns**(char)

> **Description** Returns the first character.

max_value = **proc**() **returns**(char)

> **Description** Returns the last character.

to_lower = **proc**(c: char) **returns**(char)

> **Description** Returns lower case form of c if 'A' $\leq c \leq$ 'Z'. Otherwise returns its argument.

to_upper = **proc**(c: char) **returns**(char)

> **Description** Returns upper case form of c if 'a' $\leq c \leq$ 'z'. Otherwise returns its argument.

is_lower = **proc**(c: char) **returns**(bool)

> **Description** Returns **true** if c is a lower case letter.

is_upper = **proc**(c: char) **returns**(bool)

> **Description** Returns **true** if c is a upper case letter.

is_digit = **proc**(c: char) **returns**(bool)

Description Returns **true** if c is a digit.

alpha_index = **proc**(c: char) **returns**(int) **signals**(not_alpha)

Description Returns index of the letter c.
Signals not_alpha, if c is not an upper or lower case letter.

digit_index = **proc**(c: char) **returns**(int) **signals**(not_digit)

Description Returns index of the digit c.
Signals not_digit, if c is not a digit.

make_alpha = **proc**(i: int) **returns**(char) **signals**(overflow)

Description Returns the upper case letter corresponding index i.
Signals overflow, if $i \notin \{0, ..., 25\}$.

make_digit = **proc**(i: int) **returns**(char) **signals**(overflow)

Description Returns the digit corresponding index i.
Signals overflow, if $i \notin \{0, ..., 9\}$.

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

Description Yields all characters from f to t.
Sugared form of invocation is `[f .. t]`

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

Description Yields all characters from f down to t.

lt = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1$ is less than $c2$.
Sugared form of invocation is c1 `<` c2

le = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1$ is less than or equal to $c2$.
Sugared form of invocation is c1 `<=` c2

ge = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1$ is greater than or equal to $c2$.
Sugared form of invocation is c1 `>=` c2

gt = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1$ is greater than $c2$.
Sugared form of invocation is c1 > c2

equal = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1 = c2$.
Sugared form of invocation is c1 = c2

similar = **proc**(c1, c2: char) **returns**(bool)

Description Returns **true** if $c1 = c2$.

copy = **proc**(c: char) **returns**(char)

Description Returns its argument.

to_byte = **proc**(c: char) **returns**(byte)

Description Returns the byte corresponding to c.

to_int = **proc**(c: char) **returns**(int)

Description Returns the int corresponding to c.

to_string = **proc**(c: char) **returns**(string)

Description Returns the string $\langle c \rangle$.

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

Description Returns the character corresponding to literal s.
Signals bad_format if s is not a legal character literal.

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

Description Produces a string literal corresponding to x.

A.8 Built-in Type STRING

Objects of type `string` are immutable. Each string s represents a sequentially indexed tuple of characters $\langle s_0, ..., s_{high} \rangle$. The indices will be denoted by $0, ..., high_s$. The size $size$ of a string is defined as the number of characters in it ($= high + 1$). The tuple can also be empty, i.e., $size = 0$. For any string, $size$

must be a legal int-object. Any attempts to create a string in violation of this rule results in a **failure** exception.

An object of type `string` can be created by *a string literal* "$c_1 c_2 ... c_k$", described in Section 2.5.

Operations

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

Description	Returns its argument (useful in **where**-clauses).
Sugared	form of invocation is the string literal.

fill = **proc**(n: int, c: char) **returns**(string) **signals**(negative_size)

Description	Returns the string $\langle s_0, ..., s_{n-1} \rangle$, where $s_i = c$ for all indexes i.
Signals	negative_size if $n < 0$.

concat = **proc**(s1, s2: string) **returns**(string)

Description	Returns the string $\langle s1_0, ..., s1_{high_{s1}}, s2_0, ..., s2_{high_{s2}} \rangle$.
Sugared	form of invocation is s1 || s2.

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

Description	Returns the *size* of the string s.

part = **proc**(s: string, low, count: int) **returns**(string)
signals(bounds, negative_size)

Description	Returns the string $\langle s_{low}, ..., s_{low+count-1} \rangle$, or the empty string if $count = 0$.
Signals	negative_size if $count < 0$, bounds if $low < 0$ or $low + count - 1 > high_s$.

rest = **proc**(s: string, i: int) **returns**(string) **signals**(bounds)

Description	If $i = size_s$ returns the empty string, otherwise returns the string $\langle s_i, ..., s_{high} \rangle$.
Signals	bounds if $i \notin \{0, ..., size_s\}$.

fetch = **proc**(s: string, i: int) **returns**(char) **signals**(bounds)

Description	Returns s_i.
Signals	bounds if $i \notin \{0, ..., high_s\}$.
Sugared	form of invocation is `s[i]`.

append = **proc**(s: string, c: char) **returns**(string)

Description Returns the string $\langle s_0, ..., s_{high}, c\rangle$.

map = **proc**(s: string, f: **proctype**(char) **returns**(char) **signals**(*))
returns(string) **signals**(not_possible(string))

Description Returns the string $\langle f(s_{low}), ..., f(s_{high})\rangle$.
Signals not_possible(x), if any invocation of f signals; x is name of the signal.

elements = **iter**(s: string) **yields**(char)

Description Yields, in increasing order of indexes, all characters of s.

elements_down = **iter**(s: string) **yields**(char)

Description Yields, in decreasing order of indexes, all characters of s.

indexes = **iter**(s: string) **yields**(int)

Description Yields, in increasing order, the indexes of s.

indexes_down = **iter**(s: string) **yields**(int)

Description Yields, in decreasing order, the indexes of s.

lt = **proc**(s, t: string) **returns**(bool)

Description Returns **true** if $s1$ is lexicographically less than $s2$.
Sugared form of invocation is s1 `<` s2.

le = **proc**(s1, s2: string) **returns**(bool)

Description Returns **true** if $s1$ is lexicographically less than $s2$ or equal.
Sugared form of invocation is s1 `<=` s2.

ge = **proc**(s1, s2: string) **returns**(bool)

Description Returns **true** if $s1$ is lexicographically greater than $s2$ or equal.
Sugared form of invocation is s1 `>=` s2.

gt = **proc**(s1, s2: string) **returns**(bool)

Description Returns **true** if $s1$ is lexicographically greater than $s2$.
Sugared form of invocation is s1 > s2.

equal = **proc**(s1, s2: string) **returns**(bool)

Description Returns **true** if $s1 = s2$.
Sugared form of invocation is s1 = s2.

similar = **proc**(s1, s2: string) **returns**(bool)

Description Returns **true** if $s1 = s2$.

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

Description Returns its argument.

from_array = **proc**(a: array[char]) **returns**(string)

Description Returns the string containing the same characters as a.

to_array = **proc**(s: string) **returns**(array[char])

Description Places the characters of the argument as elements of a new array.

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

Description Returns the string corresponding to string literal.
Signals bad_format if s is not a legal string literal.

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

Description Produces the string literal corresponding to s.

A.9 Built-in Type Generator ARRAY

Objects of types generated by `array` are one-dimensional mutable containers. For each type `T` there exists a type `array[T]`. The state of an object a of type `array[T]` consists of a sequentially indexed tuple of elements $\langle a_0, a_1, ..., a_{high} \rangle$, where all elements are of type `T`. The size $size$ of an array is defined as the number of elements in the tuple, that can also be empty. The high bound $high$ of the array is $size - 1$. For any array, $size$ must be a legal `int`-object. Any attempt to create or modify an array in violation of this rule results in a **failure** exception.

The *size* of objects of type `array[T]` is not predefined, but after creation of an array object its *size* remains unchanged (static).

All arrays of type `array[T]` (regardless of their sizes) are assignment compatible with variables of the same type.

Arrays use the sequence literal $[e_1, e_2, ..., e_k]$ described in Section 2.5.

Operations

new = **proc**() **returns**(array[t])

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

alloc = **proc**(n: int) **returns**(array[t])

> **Description** Returns a new uninitialized array, whose size is n.

create = **proc**(s: sequence[t]) **returns**(array[t])

> **Description** Returns a new array $\langle s_0, ..., s_{high}\rangle$, where $s_i = s[i]$.
> **Sugared** form of invocation is $[s_0, ..., s_{high}]$.

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

> **Description** Returns a new array $\langle a_0, ..., a_{n-1}\rangle$, where all elements a_i share the object e.
> **Signals** Signals negative_size if $n < 0$.

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

> **Description** Returns a new array $\langle a_0, ..., a_{n-1}\rangle$, where elements are copies of e.
> **Signals** Signals negative_size if $n < 0$.

size = **proc**(a: array[t]) **returns**(int)

> **Description** Returns *size*.

part = **proc**(a: array[t], lb, n: int) **returns**(array[t])
signals(bounds, negative_size)

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

fetch = **proc**(a: array[t], i: int) **returns**(t) **signals**(bounds)

Description Returns the element a_i of a.
Signals bounds if $i \notin \{0, ..., high_a\}$.
Sugared form of invocation is a[i].

store = **proc**(a: array[t], i: int, x: t) **signals**(bounds)

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

map = **proc**(a: array[t], f: **proctype**(t) **returns**(t) **signals**(*))
returns(array[t]) **signals**(not_possible(string))

Description Returns a new array $\langle f(s_{low}), ..., f(s_{high})\rangle$.
Modifies Can have secondary side-effects on a due to f.
Signals not_possible(x), if any invocation of f signals; x is name of the signal.

elements = **iter**(a: array[t]) **yields**(t)

Description Yields all elements a_i in increasing order of indexes.

elements_down = **iter**(a: array[t]) **yields**(t)

Description Yields all elements a_i in decreasing order of indexes.

indexes = **iter**(a: array[t]) **yields**(int)

Description Yields all indexes of an array in increasing order.

indexes_down = **iter**(a: array[t]) **yields**(int)

Description Yields all indexes of an array in decreasing order.

equal = **proc**(a1, a2: array[t]) **returns**(bool)

Description Returns **true** if both arguments are the same object.
Sugared form of invocation is a1 = a2.

similar = **proc**(a1, a2: array[t]) **returns**(bool)
where t **has** similar: **proctype**(t, t) **returns**(bool)

Description Returns **true** if t$similar($a1_i, a2_i$) is **true** for all indexes i.
Modifies Can have secondary side-effects on $a1$ and $a2$ due to t$similar.

equal_state = **proc**(a1, a2: array[t]) **returns**(bool)
where t **has** equal: **proctype**(t, t) **returns**(bool)

Description Returns **true** if $a1$ and $a2$ have the same state.
Modifies Can have secondary side-effects on $a1$ and $a2$ due to t$equal.

copy = **proc**(a: array[t]) **returns**(array[t])
where t **has** copy: **proctype**(t) **returns**(t)

Description Returns a copy of a, $\langle e_0, ..., e_{high_a} \rangle$, where e_i = t$copy(a_i) for all indexes i.
Modifies Can have secondary side-effects on a due to t$copy.

copy_state = **proc**(a: array[t]) **returns**(array[t])

Description Creates a new array with the same state as a.

to_sequence = **proc**(a: array[t]) **returns**(sequence[t])

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

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

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

unparse = **proc**(a: array[t]) **returns**(string)
where t **has** unparse: **proctype**(t) **returns**(string)

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

A.10 Built-in Type Generator SEQUENCE

Objects of types generated by `sequence` are one-dimensional immutable containers. For each type `T` there exists a type `sequence[T]`. Each object of type `sequence[T]` represents a sequentially indexed tuple of elements $\langle s_0, s_1, ..., s_{high} \rangle$, where all elements are of type `T`. The size *size* of a sequence is defined as the number of elements in the tuple, that can also be empty. For any sequence, *size* must be a legal `int`-object. Any attempt to create or modify a sequence in violation of this rule results in a **failure** exception.

The *size* of objects of type `sequence[T]` is not predefined; therefore all sequences of type `sequence[T]` (regardless of their sizes) are assignment compatible with variables of the same type.

Sequences use the sequence literal form `[`$e_1, e_2, ..., e_k$`]` discussed in Section 2.5.

Operations

new = **proc**() **returns**(sequence[t])

Description Returns the empty sequence.

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

Description Returns its argument (useful in **where**-clauses).
Sugared form of invocation is `[`$s_0, ..., s_{high}$`]`.

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

Description Returns the sequence $\langle s_0, ..., s_{n-1} \rangle$, where $s_i = e$ for all indexes i.
Signals negative_size if $n < 0$.

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

Description Returns the sequence $\langle s_0, ..., s_{n-1} \rangle$, where s_i = t\$copy(e) for all indexes i.
Signals negative_size if n < 0.

concat = **proc**(s1, s2: sequence[t]) **returns**(sequence[t])

Description Returns the concatenation of of $s1$ and $s2$.
Sugared form of invocation is s1 || s2 .

size = **proc**(s: sequence[t]) **returns**(int)

Description Returns *size* of s.

part = **proc**(s: sequence[t], lb, n: int) **returns**(sequence[t])
signals(bounds, negative_size)

Description Returns the sequence $\langle s_{lb}, ..., s_{lb+n-1} \rangle$, or the empty sequence if $n = 0$.
Signals negative_size if $n < 0$, bounds if $lb < 0$ or $lb+n-1 > size_s$.

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

Description Returns the element s_i.
Signals bounds if $i \notin \{0, ..., high_a\}$.
Sugared form of invocation is s[i]

replace = **proc**(s: sequence[t], i: int, e: t) **returns**(sequence[t]) **signals**(bounds)

Description Returns copy of s, where s_i has been replaced with e.
Signals bounds if $i \notin \{0, ..., high_a\}$.

map = **proc**(s: sequence[t], f: **proctype**(t) **returns**(t) **signals**(*))
returns(sequence[t]) **signals**(not_possible(string))

Description Returns the sequence $\langle f(s_{low}), ..., f(s_{high}) \rangle$.
Signals not_possible(x), if any invocation of f signals; x is name of the signal.

elements = **iter**(s: sequence[t]) **yields**(t)

Description Yields all elements s_i in increasing order of indexes.

elements_down = **iter**(s: sequence[t]) **yields**(t)

Description Yields all elements s_i in decreasing order of indexes.

indexes = **iter**(a: sequence[t]) **yields**(int)

Description Yields all indexes of a sequence in increasing order.

indexes_down = **iter**(a: sequence[t]) **yields**(int)

Description Yields all indexes of a sequence in decreasing order.

equal = **proc**(s1: sequence[t], s2:sequence[t]) **returns**(bool)
where t **has** equal: **proctype**(t, t) **returns**(bool)

Description	Returns **true** if *s1* and *s2* have pairwise equal elements as determined by t$equal.
Modifies	Can have secondary side-effects on $s1$ and $s2$ due to t$equal.
Sugared	form of invocation is s1 = s2

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

Description	Returns **true** if s1 and s2 have pairwise similar elements as determined by t$similar.
Modifies	Can have secondary side-effects on $s1$ and $s2$ due to t$similar.

copy = **proc**(s: sequence[t]) **returns**(sequence[t])
where t **has** copy: **proctype**(t) **returns**(t)

Description	Returns the sequence $\langle r_0, ..., r_{high_s} \rangle$, where r_i = t$copy(s_i).
Modifies	Can have secondary side-effects on s due to t$copy.

to_array = **proc**(s: sequence[t]) **returns**(array[t])

Description	Returns a new array that has the same elements as s.

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

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

unparse = **proc**(s: sequence[t]) **returns**(string)
where t **has** unparse: **proctype**(t) **returns**(string)

Description	Returns a string containing the elements of s unparsed by t$unparse in the form `[` *element* {, *element* } `]`.
Modifies	Can have secondary side-effects on s due to t$unparse.

A.11 Built-in Type Generator TAG

Tag generates types that model simple enumerated labels. Objects of types generated by tag are immutable.

For every field list F = {$N_1 : T_1, ..., N_k : T_k$} there exists a type `tag{F}`. The order of the fields is significant, i.e., for each permutation of fields there exists a distinct type. The field list defines a set of names (tags) (the types have no significance). Each object of type `tag{F}`, represents one of these names.

The tag literal 'N_i is described in Section 2.5.

Operations

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

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

count = **proc**() **returns**(int)

Description	Returns the number of fields (tags) that belong to the type.

is_N_i = **proc**(t: tag{F}) **returns**(bool)

Description	Returns **true** if the tag t is N_i. There is an is_N_i operation for each field.

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

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

tags = **iter**() **yields**(tag{F})

Description	Yields all tags in their definition order.

tags_down = **iter**() **yields**(tag{F})

Description	Yields all tags in reversed definition order.

equal = **proc**(t1, t2: tag{F}) **returns**(bool)

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

similar = **proc**(t1, t2: tag{F}) **returns**(bool)

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

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

Description Returns its argument.

from_int = **proc**(i: int) **returns**(tag{F}) **signals**(bad_tag)

Description Returns the tag corresponding to the given field index.
Signals bad_tag if there is no tag with index i.

to_int = **proc**(t: tag{F}) **returns**(int)

Description Returns the index of tag t.

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

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

unparse = **proc**(t: tag{F}) **returns**(string)

Description Returns a string representing tag t.

A.12 Built-in Type Generator VARIANT

Variant generates types that are discriminating unions of tagged types and tags. Objects of types generated by `variant` are mutable and model tags and tagged objects.

For every field list F = {$N_1 : T_1, ..., N_k : T_k$} there exists a type `variant{F}`. The fields may be written in arbitrary order. The state of an object of type `variant{F}`, is a *tag* N_i or a *tag/value pair* (N_i, e_i), where e_i is an object of type T_i.

Variants can use the tag and oneof literal forms described in Section 2.5.

Operations

create_N_i = **proc**(e: T_i) **returns**(variant{F})

Description Returns a new variant whose state is the pair (N_i, e). There is a create_N_i operation for each field.
Sugared form of invocation is 'N_i(e)

is_N_i = **proc**(v: variant{F}) **returns**(bool)

Description Returns **true** if the tag of v is N_i. There is an is_N_i operation for each field.

get_N_i = **proc**(v: variant{F}) **returns**(T_i) **signals**(wrong_tag)

Description Returns the value of tag/value pair v.
Signals wrong_tag if the tag of v is other than N_i.
Sugared form of invocation is v.N_i

set_N_i = **proc**(v: variant{F}, e: T_i)

Description Changes the state of v to be the pair (N_i, e). There is a set_N_i operation for each field.
Modifies state of v.
Sugared form of invocation is v.N_i := e

tag_of = **proc**(v: variant{F}) **returns**(tag{F})

Description Returns the corresponding tag.

equal = **proc**(v1, v2: variant{F}) **returns**(bool)

Description Returns **true** if $v1$ and $v2$ are the same object.
Sugared form of invocation is v1 = v2.

similar = **proc**(v1, v2: variant{F}) **returns**(bool)
where T_i **has** similar: **proctype**(T_i, T_i) **returns**(bool)

Description If $v1$ and $v2$ have different tags, the result is false. Otherwise the result is that of invoking T_i\$similar with the two value components.
Modifies Can have secondary side-effects on $v1$ and $v2$ due to t\$similar.

equal_state = **proc**(v1, v2: variant{F}) **returns**(bool)
where T_i **has** equal: **proctype**(T_i, T_i) **returns**(bool)

Description Returns **true** if $v1$ and $v2$ have the same state.
Modifies Can have secondary side-effects on $v1$ and $v2$ due to t\$equal.

copy = **proc**(v: variant{F}) **returns**(variant{F})
where T_i **has** copy: **proctype**(T_i) **returns**(T_i)

Description If the state of v is (N_i, e), then the result is a new variant whose initial state is (N_i, T_i\$copy(e)). Otherwise state of the result is N_i.
Modifies Can have secondary side-effects on v due to t\$copy.

copy_state = **proc**(v: variant{F}) **returns**(variant{F})

Description Returns a new variant with the same state as v has.

copy_oneof = **proc**(v: variant{F}, o: oneof{F})

Description Changes v to have the same state as o.

copy_variant = **proc**(v1, v2: variant{F})

Description Changes $v1$ to have the same state as $v2$.

to_oneof = **proc**(v: variant{F}) **returns**(oneof{F})

Description Returns the corresponding oneof.

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

Description Parses a string of syntax '*tag*(*element*) where *tag* is one of N_i and *element* is represented in the format of T_i\$parse. If type of element is `null` then the form '*tag* can be used. Returns a new variant corresponding to s.
Signals syntax_error(m, i) where m is the error message and i the position in s.

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

Description Returns a string of the form '*tag*(*element*) where *tag* is N_i and the *element* is represented in the format of T_i\$unparse. If T_i is `null` then the form '*tag* is used.
Modifies Can have secondary side-effects on v due to t\$unparse.

A.13 Built-in Type Generator ONEOF

Oneof generates types that are discriminating unions of tagged types and tags. Objects of types generated by `oneof` are immutable and model tags and tagged objects.

For every field list F = `{`$N_1 : T_1, ..., N_k : T_k$`}` there exists a type `oneof{F}`. The fields may be written in arbitrary order. Each object of type `oneof{F}`, represents a *tag* N_i or *tag/value pair* (N_i, e_i), where e_i is an object of type T_i.

Oneofs can use the tag and oneof literal forms described in Section 2.5.

Operations

create_N_i = **proc**(e: T_i) **returns**(oneof{F})

Description	Returns the oneof representing the pair (N_i, e). There is a create_N_i operation for each field.
Sugared	form of invocation is ‘N_i(e)

is_N_i = **proc**(o: oneof{F}) **returns**(bool)

Description	Returns **true** if the tag of *o* is N_i. There is an is_N_i operation for each field.

get_N_i = **proc**(o: oneof{F}) **returns**(T_i) **signals**(wrong_tag)

Description	Returns the value of tag/value pair *o*. There is a get_N_i operation for each field.
Signals	wrong_tag if the tag of *o* is other than N_i.
Sugared	form of invocation is o.N_i

tag_of = **proc**(o: oneof{F}) **returns**(tag{F})

Description	Returns the corresponding tag.

equal = **proc**(o1, o2: oneof{F}) **returns**(bool)
where T_i **has** equal: **proctype**(T_i, T_i) **returns**(bool)

Description	If *o*1 and *o*2 have different tags, the result is false. Otherwise result is that of invoking T_i\$equal with the two value components.
Modifies	Can have secondary side-effects on the arguments due to T_i\$equal.
Sugared	form of invocation is o1 `=` o2.

similar = **proc**(o1, o2: oneof{F}) **returns**(bool)
where T_i **has** similar: **proctype**(T_i, T_i) **returns**(bool)

Description If $o1$ and $o2$ have different tags, the result is false. Otherwise result is that of invoking T_i\$similar with the two value components.

Modifies Can have secondary side-effects on the arguments due to T_i\$similar.

copy = **proc**(o: oneof{F}) **returns**(oneof{F})
where T_i **has** copy: **proctype**(T_i) **returns**(T_i)

Description If the state of o is (N_i, e), then the result is a new oneof whose initial state is (N_i, T_i\$copy(e)). Otherwise the result is N_i.

Modifies Can have secondary side-effects on o due to T_i\$copy.

to_variant = **proc**(o: oneof{F}) **returns**(variant{F})

Description Returns the corresponding variant.

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

Description Parses a string of the syntax '*tag*(*element*) where *tag* is one of N_i and *element* is represented in the format of T_i\$parse. If type of the element is `null` then the form '*tag* can be used. Returns the oneof corresponding to s.

Signals syntax_error(m, i) where m is the error message and i the position in s.

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

Description Returns a string of the form '*tag*(*element*) where *tag* is N_i of o and the *element* is represented in the format of T_i\$unparse. If T_i is `null` then the form '*tag* is used.

Modifies Can have secondary side-effects on o due to T_i\$unparse.

A.14 Built-in Type Generator RECORD

Objects of types generated by `record` are mutable and model heterogeneous sets of tagged objects.

For every field list F = {N_1 : $T_1, ..., N_k$: T_k} there exists a type `record{F}`. The fields may be written in arbitrary order. The state of an object r of type `record{F}`, is a set of tagged elements $\{r_{N_1}, ..., r_{N_k}\}$, where each element r_{N_i} is of type T_i.

Records use the struct literal {$e_1, e_2, ..., e_k$} described in Section 2.5.

Operations

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

default = **proc**() **returns**(record{F})
where T_i **has** default: **proctype**() **returns**(T_i)

Description	Returns a new record filled with default objects of each component type.

get_N_i = **proc**(r: record{F}) **returns**(T_i)

Description	Returns r_{N_i}. There is a get_N_i operation for each field.
Sugared	form of invocation is r.N_i .

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

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

equal = **proc**(r1, r2: 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: 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 secondary side-effects on $r1$ and $r2$ due to the T_i\$similars.

equal_state = **proc**(r1, r2: 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 secondary side-effects on $r1$ and $r2$ due to the T_i\$equals.

copy = **proc**(r: record{F}) **returns**(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 secondary side-effects on r due to the T_i\$copy operations.

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

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

copy_record = **proc**(r1, r2: record{F})

Description Changes the state of $r1$ to have the same components as $r2$.

copy_struct = **proc**(r: record{F}, s: struct{F})

Description Changes the state of r to have the same components as s.

to_struct = **proc**(v: record{F}) **returns**(struct{F})

Description Returns the corresponding struct.

parse = **proc**(s: string) **returns**(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* {, *tag*: *element*}} where each *tag* is some N_i and the corresponding *element* is represented in the format of T_i\$parse. Returns a new record corresponding to s.
Signals syntax_error(m, i) where m is the error message and i the position in s.

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

Description Returns a string $\{N_1 : element_1, ..., N_k : element_k\}$ where $elements$ are represented in the format of T_i`$`unparse.
Modifies Can have secondary side-effects on v due to t`$`unparse.

A.15 Built-in Type Generator STRUCT

Objects of types generated by `struct` are immutable and model heterogeneous sets of tagged objects.

For every field list F = $\{N_1 : T_1, ..., N_k : T_k\}$ there exists a type `struct{F}`. where the fields may be written in arbitrary order. Each object s of type `struct{F}`, represents a set of tagged elements $\{s_{N_1}, ..., s_{N_k}\}$, where each element s_{N_i} is of type T_i.

Structs use the struct literal $\{e_1, e_2, ..., e_k\}$ described in Section 2.5.

Operations

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

Description Returns its argument (useful in **where**-clauses).
Sugared form of invocation is the struct literal.

default = **proc**() **returns**(struct{F})
where T_i **has** default: **proctype**() **returns**(T_i)

Description Returns a new struct filled with default objects of each component type.

get_N_i = **proc**(s: struct{F}) **returns**(T_i)

Description Returns s_{N_i}. There is a get_N_i operation for each field.
Sugared form of invocation is s.N_i .

replace_N_i = **proc**(s: struct{F}, a: T_i)

Description Returns a copy of s where the N_i-component is replaced with a. There is a replace_N_i operation for each field.

equal = **proc**(s1, s2: struct{F}) **returns**(bool)
where T_i **has** equal: **proctype**(T_i, T_i) **returns**(bool)

Description Returns **true** the components of $s1$ and $s2$ are pairwise equal.
Modifies Can have secondary side-effects on the arguments due to T_i\$equal.
Sugared form of invocation is s1 = s2.

similar = **proc**(s1, s2: struct{F}) **returns**(bool)
where T_i **has** similar: **proctype**(T_i, T_i) **returns**(bool)

Description For each i 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 secondary side-effects on the arguments due to T_i\$similar.

copy = **proc**(s: struct{F}) **returns**(struct{F})
where T_i **has** copy: **proctype**(T_i) **returns**(T_i)

Description Returns the struct $\langle T_1\$copy(s_{N_1}), ..., T_k\$copy(s_{N_k})\rangle$.
Modifies Can have secondary side-effects on s due to T_i\$copy.

to_record = **proc**(o: struct{F}) **returns**(record{F})

Description Returns the corresponding record.

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

Description Parses a list structure { *tag*: *element* {, *tag*: *element*}} where each *tag* is some N_i and the corresponding *element* is represented in the format of T_i\$parse. Returns the struct corresponding to s.
Signals syntax_error(m, i) where m is the error message and i the position in s.

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

Description Returns a string $\{N_1 : element, ..., N_k : element\}$ where *elements* are represented in the format of T_i\$unparse.
Modifies Can have secondary side-effects on s due to T_i\$unparse.

zap = **proc**()

Appendix B: Input/Output

B.1 General Aspects

The input and output facilities of XE programs are not bound to the language, but for better portability there exists a "standard library" for input and output. The library contains stream-like abstractions, i.e., abstractions that treat files as data structures where the elements can be accessed only sequentially one after another.

These abstractions are somewhat machine dependent, because the file system and I/O-system calls can differ a great deal from one operating system to another. The types are expected to have at least the operations listed below.

B.2 Type STREAM

Stream is an abstraction of sequentially accessible files of characters. Objects of type stream are meant to be used as the normal channel for textual input and output, and are mutable.

Some operations of stream are dependent on the target machine and its operating system.

Type `stream_mode` is typically `tag{read, write, append:  null}`

Operations

open = **proc**(f: filename, mode: stream_mode) **returns**(stream)
signals(not_possible(string))

Description	Opens a file in given mode and returns the corresponding stream.
Signals	not_possible(s) if error occurs, s is a description of the error.

primary_input = **proc**() **returns**(stream)

Description	Returns a stream corresponding to primary input.

primary_output = **proc**() **returns**(stream)

Description	Returns a stream corresponding to primary output.

error_output = **proc**() **returns**(stream)

Description Returns a stream corresponding to error output.

close = **proc**(s: stream)

Description Closes s.

is_output = **proc**(s: stream) **returns**(bool)

Description Returns **true** if characters can be written into s.

is_input = **proc**(s: stream) **returns**(bool)

Description Returns **true** if characters can be read from s.

getc = **proc**(s: stream) **returns**(char) **signals**(end_of_file, not_possible(string))

Description Reads the next character from s.
Modifies state of s.
Signals end_of_file if there are no characters left in s, not_possible(string) otherwise if the next character cannot be read.

peekc = **proc**(s: stream) **returns**(char) **signals**(end_of_file, not_possible(string))

Description Returns the next character from s. Does not advance the current read position in the stream.
Signals end_of_file if there are no characters left in s, not_possible(string) otherwise if the next character cannot be read.

putc = **proc**(s: stream, c: char) **signals**(not_possible(string))

Description Writes a character to s.
Modifies state of s.
Signals not_possible(s) when writing is not possible.

getl = **proc**(s: stream) **returns**(string) **signals**(end_of_file, not_possible(string))

Description Reads the next the line from s, i.e. reads until next new line characters is found and skips it.
Modifies state of s.
Signals end_of_file if there are no characters left in s, not_possible(string) otherwise if the next character cannot be read.

getn = **proc**(s: stream, n: int) **returns**(string) **signals**(eof_reached(string))

Description	Reads n characters from s.
Modifies	state of s.
Signals	eof_reached(r) if there are less than n characters left in s, r contains them.

gets = **proc**(s: stream) **returns**(string) **signals**(end_of_file, not_possible(string))

Description	Skips potential initial white space and thereafter reads characters from s until a white space character is met. It is not included in the string returned. Characters ' ', \n and \t are considered white space.
Modifies	state of s.
Signals	end_of_file if there are no characters left in s.

putl = **proc**(s: stream, c: string)

Description	Writes a line to s.
Modifies	state of s.

puts = **proc**(s: stream, str: string)

Description	Writes a string to s.
Modifies	state of s.

eof = **proc**(s: stream) **returns**(bool)

Description	Returns **true** if there are no characters left in s.

is_closed = **proc**(s: stream) **returns**(bool)

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

reset = **proc**(s: stream)

Description	Set the first character of s to be the next one, i.e. rewinds the file and starts from the beginning.
Modifies	state of s.

flush = **proc**(s: stream)

Description	Flushes all buffered output.
Modifies	state of s.

equal = **proc**(s1, s2: stream) **returns**(bool)

Description Returns **true** if $s1 = s2$.
Sugared form of invocation is s1 `=` s2.

similar = **proc**(s1, s2: stream) **returns**(bool)

Description Returns **true** if $s1 = s2$.

copy = **proc**(s: stream) **returns**(stream)

Description Returns its argument.

B.3 Type BINARY

Binary is an abstraction of sequentially accessible machine-level (binary) data. Objects of type binary are mutable and meant to be used for communication with systems that are not written in XE, such as the file system, or message passing facilities.

Some operations of binary are dependent on the target machine and its operating system.

Type byte_mode is typically `tag{read, write:  null}`

Binary is available only for those types T, that have `dump` and `undump`-operations (all built-in types have) as described in Section 6.3.

Operations

create_stream = **proc**(s: stream, m: binary_mode) **returns**(binary)
signals(not_possible(string))

Description Creates a binary in mode m that reads from (writes to) the given stream.
Signals not_possible(s) if error occurs, s is a description of the error.

create_dynamic = **proc**(m: binary_mode) **returns**(binary)

Description Creates a binary in mode m that reads from (writes to) a dynamic memory area.

create_memory = **proc**(a: array[byte], m: binary_mode) **returns**(binary)
signals(not_possible(string))

Description	Creates a binary in mode m that reads from (writes to) the array a.
Signals	not_possible(s) if error occurs, s is a description of the error.

get_byte = **proc**(b: binary) **returns**(byte) **signals**(end_of, not_possible(string))

Description	Reads one byte from b.
Modifies	state of b
Signals	end_of, if there are no bytes left in b; not_possible(s) if reading is not possible.

peek_byte = **proc**(b: binary) **returns**(byte) **signals**(end_of, not_possible(string))

Description	Reads one byte from b, but does not advance the current read position of b.
Modifies	state of b
Signals	end_of, if there are no bytes left in b; not_possible(s) if reading is not possible.

put_byte = **proc**(b: binary, d: byte) **signals**(end_of, not_possible(string))

Description	Reads one byte from b.
Modifies	state of b
Signals	end_of, if there are no bytes left in b; not_possible(s) if reading is not possible.

is_output = **proc**(b: binary) **returns**(bool)

Description	Returns **true** if objects can be written into b.

is_input = **proc**(b: binary) **returns**(bool)

Description	Returns **true** if objects can be read from b.

empty = **proc**(b: binary) **returns**(bool)

Description	Returns **true** if there are no objects left in b.

reset = **proc**(b: binary, m: binary_mode)

Description Rewinds the binary and starts from the beginning in mode m.
Modifies state of b.

equal = **proc**(b1, b2: binary) **returns**(bool)

Description Returns **true** if $b1 = b2$.
Sugared form of invocation is b1 = b2.

similar = **proc**(b1, b2: binary) **returns**(bool)

Description Returns **true** if $b1 = b2$.

copy = **proc**(b: binary) **returns**(binary)

Description Returns its argument.