Foreword

XE is a general purpose language that is based on the concept of abstract data types. XE owes much to Clu, which has been used as a starting point in its design. The mechanisms for defining procedures, iterators and data types will be familiar to anybody knowing Clu.

This manual is an introduction to the XE programming language. It is meant for learning the language and the XE programming style. Basic concepts and features of XE are introduced by means of examples. This manual does not contain a detailed description of XE. The XE language is thoroughly described in the *XE Reference Manual*, which is useful for any reader of this manual.

To use the XE language you should be familiar to the XE programming environment. To learn to use the environment properly, you should read the *XE Programming Environment User's Guide*.

Acknowledgements

The programming language XE has been designed and implemented at the Helsinki University of Technology within the ExBed project (Expert System Framework for Embedded Applications) funded by the Technology Development Centre, Nokia Corporation and KONE Corporation.

1. Introduction

1.1 Modules

An XE program is made up of one or more *modules*. Each module is a *procedure*, *iterator*, *rule*, *data type*, or *constant*. Procedures, iterators, and rules are called *routines*, because they can be invoked. There exists no concept of a main program, because any routine that can be directly invoked by the execution environment can act as a main program. For example, the following simple procedure is an XE program:

```
echo = proc()
    input: stream := stream$primary_input()
    output: stream := stream$primary_output()
    puts(:output, "Give a line : ")
    line: string := getl(:input)
    putl(:output, "The line you just wrote was " || line)
end
```

The program reads a line from the primary input device and outputs it to the primary output device (typically the user's terminal or console).

Constants are simple definitions like:

```
max_size = 12                    % note that = is used instead of :=
table_lenght = max_size - 5
```

Procedures can have arguments and they can return result values. The `divmod` below procedure returns two values: the result of dividing its first argument by the second one and the remainder of this division.

```
divmod = proc(x, y: int) returns(int, int)
    quotient: int := x / y
    remainder: int := x // y
    return(quotient, remaider)
end
```

Iterators resemble procedures. When a procedure is invoked, it returns the result values *after* the termination of its activation. An iterator can yield result values several times *during* its activation.

Rules are used to implement a condition-action abstraction. A rule is a special kind of an iterator. It is given zero or more argument objects and it creates a sequence of items, that it yields one at a time. Usually they contain data that satisfies the *condition* of the rule.

A data type implements data abstraction. It defines a set of objects and the operations allowed to manipulate those objects. The operations can be procedures, iterators, or rules. A typical data type is `int` whose objects belong to a subrange of the mathematical integers and whose operations form the usual integer arithmetics.

Lexically an XE module consists of a sequence of *tokens* (symbols). Tokens can be words, operators (`+`, `*`, `$`...), *literals*(2,′c′,$...), *orpunctuationmarks*(@,$,$...).*Awordisasequenceofletters*, *digitsandunderscoresthatbeginswithaletteroranunderscore.Someofthewordsarereserved*, *becausetheyareusedinconstructsofthelanguage.Reservedwordsarewordslike***begin**, **end**, *and***if**.(*Thecompletesyntax*, *includinglistofreservedwords*, *operatorsandpunctuationmarkscanbefoundintheXEReferenceManual*.)

Tokens can be separated by separators: blanks, horizontal tabs or new lines. An XE module can also contain *comments*. A comment starts with `%` and continues to the end of the line. Separators and comments do not affect the behavior of a module.

Below is an example routine that returns true if all characters in the argument string *s* belong to the alphabet. Characters of the argument string are accessed using the iterator `elements` of the type `string`. (The colon notation in `elements(:s)` is used to denote this. Another way would be to write `string$elements`.)

```
string_is_alpha = proc(s: string) returns(bool)

  % A local procedure visible only here

  char_is_alpha = proc(c: char) returns(bool)
    return('A' <= c cand c <= 'Z' cor 'a' <= c cand c <= 'z')
  end

  for c: char in elements(:s) do
    if ~char_is_alpha(c) then      % ~ is the not operation
      return(false)
    end
  end
  return(true)
end
```

As in the previous example, modules can have a nested structure relative to one another. For instance, the operations of a data type are implemented by nested routines.

1.2 Objects and Variables

The basic concepts of XE are *objects* and *variables*. XE programs refer to data represented by objects and perform computations on them. Variables are names that are used to refer to objects. A reference is established by an assignment.

```
a := 5
```

After this assignment, the variable *a* refers to an object having 5 as its value.

Each object and each variable has a type. The type of an object completely determines what kind of values an object may have and what operations can be performed on it. For example, integer addition operation is meaningful for integer objects but not for string objects. An object can be assigned only to a variable of a compatible type. Except for some special cases (routine objects) this means that the type of the object and the type of the variable must be exactly the same. There exist no implicit conversions in XE. So, the second assignment in the following example is *illegal*, even though there exists an integer object having value 5.

```
b: byte, i: int
b := 5                  % 5 represents a byte, because b is of the type byte
i := b
```

In XE variables must be declared before they are used. The scope of a variable is from its declaration to the end of the smallest scoping unit containing its declaration excluding contained submodules that do not bind it (binding is explained in Section 6). Scoping units correspond to the nesting structure of statements and modules.

```
scope_example = proc(i: int) returns(ityp)
    ityp = int          % A constant can used before declared, but a variable not
    a: int              % Declaration without initialization
    a := i / 5
    if a > i then
        b: int := a + 2 % Declaration with initialization
    end
    return(a)           % a is visible here, but b is not
end
```

Objects can refer to other objects. It is possible to have cyclic objects (which refer to themselves) and recursive data structures without the use of explicit pointers. For example the array

```
ar: array[int] := [1, 3, 4]
```

refers to integer objects 1, 3, and 4. (The array is usually said to *contain* the objects.)

Several objects or variables can refer to the same object. When this happens, we say that they *share* the object referred to. In the following example the array object having [1, 2] as its initial value is shared by *x* and *y*. Because of the sharing, also y[0] is 3 after the last assignment.

```
x: array[int] := [1, 2]
y: array[int] := x
x[0] := 3
```

There are two kinds of objects in XE, those that always have the same value and those whose value can change. Objects that always have the same value are called *immutable*. For example, integer objects are immutable: the integer object having 3 as its value will always have 3 as its value. Whether an object is immutable depends on the operations that its type supports. If there exist operations that can modify the objects, then the objects are *mutable*. The current value of a mutable object depends on its *state*. The mutability of an object strongly affects its abstract behavior.

For instance, arrays are mutable as can be seen in the previous example. When an object is immutable, it does not matter how many other objects share it; since it can never be modified, we cannot observe the effect of sharing. Actually, to achieve a greater efficiency, the compiler can make a copy of an immutable object instead of sharing it.

1.3 Control Structures

Within a module, XE uses conventional control structures, like the **if** and **while** statements used in figure 1.

The normal branch statement is the **if** statement. The **elseif** and **else** branches are optional. The expressions are evaluated in succession until one is found to be true. The body corresponding to that expression is executed, and this terminates the execution of the **if** statement. If no expression is true, the body in the **else** branch is executed (if it is present).

The body of the **while** statement is executed repeatedly as long as the expression remains true. The expression is evaluated before each execution of the body.

```
binary_search = proc(a: ai, x: int) returns(int)
    ai = array[int]    % note that this constant can be used in the header
    low: int := 0
    high: int := ai$size(a) - 1
    while low <= high do
        try: int := (low + high)/2
        if a[try] > x then
            high := try - 1
        elseif x > a[try] then
            low := try + 1
        else
            return(try)
        end
    end
end
```

Figure 1

The execution of a **while** or **for** loop can be terminated by a **break** statement. (The **for** loop used together with iterators is discussed in Section 5.) When a **break** statement is executed the control is transferred to the statement following the end of the smallest loop statement in which the **break** appears. The following piece of code tests if character *ch* can be found in the string *s*. The loop is broken at the first occurrence of a character equal to *ch*.

```
found: bool := false
for c: char in elements(:s) do
    if c = ch then
        found := true
        break
    end
end
```

Sometimes a **while** loop is used to perform successive operations of which some are optional. To avoid complex **if-then-elseif-** structures the **continue** statement can be used to immediately transfer control to the beginning of the loop.

The next example searches three character sequences from the input matching the pattern (left delimiter, alphabet, right delimiter) and stores the sequences found in *result*. The reader should notice that the condition of the **while** statement is evaluated after each execution of **continue** before entering the body.

```
while ˜eof(:input) do
    first: char := getc(:input)
    if ˜is_left_delim(first) cor eof(:input) then
        continue
    end
    second: char := getc(:input)
    if ˜is_alpha(second) cor eof(:input) then
        continue
    end
    third: char := getc(:input)
    if ˜is_right_delim(third) then
        continue
    end
    insert(:result, [first, second, third])
end
```

Occasionally long **if-elseif-elseif-** chains would be needed to determine the value of an object. Instead of such a structure the **case** statement should be used. The expression in the **case** heading is executed. It is compared in succession with the expressions in the **when** options and the body of the first matching one is executed. If no match is found, the **otherwise** option is executed (if it is present).

```
cmd: string := getl(:input)
case get_name(cmd) of
  when "write":
    action_write(get_arg1(cmd))
  when "read":
    action_read(get_arg1(cmd))
  otherwise:
    putl(:output, "Illegal command")
end
```

The example executes the action corresponding to the command given. `Get_name` returns the command name and `get_arg1` returns the first argument.

2. Writing Procedures

A procedure is used for implementing procedural abstraction, i.e., mappings from a set of argument objects to a set of result objects. A procedure is provided with zero or more argument objects, it returns zero or more result objects, and possibly modifies the state of its arguments. A typical procedure looks like:

```
string_similar = proc(s1, s2: string) returns(bool)
    if size(:s1) ~= size(:s2)  then              % ~= means not equal
        return(false)
    end
    for i: int in indexes(:s1) do
        if ~similar(:s1[i], s2[i]) then
            return(false)
        end
    end
    return(true)
end
```

This procedure compares two strings and returns **true** if they contain similar characters (for `chars` similarity is equality).

Each such definition has a *header* that specifies the number, order, and types of the arguments that must be passed to the procedure when it is called, and the number, order, and types of the results returned from the call.

The special statement **return** is used to return result values to the invoker. The invocation (activation) is terminated immediately when **return** is executed. The number, order, and types of the result values must match with the **returns** clause in the header.

Note that parentheses must be used even if no arguments are passed:

```
main_program()
```

If a procedure returns values, they can be assigned to variables:

```
is_similar: bool := string_similar("abc", x)
```

When the procedure is invoked, *s1* is set to refer to "abc" and *s2* to the object referred to by *x*. The argument passing mechanism is similar to assignment – the argument objects are *shared* by the invoker and procedure invoked. A procedure is able to change the objects passed to it (if they are mutable), but it cannot assign

```
binary_search = proc(a: ai, x: int) returns(int) signals(missing_from(int))
    ai = array[int]
    if ai$size(a) = 0 then
        signal missing_from(0)
    end
    low: int := 0
    high: int := ai$size(a) - 1
    if x > a[high] then
        signal missing_from(high + 1)
    elseif a[low] > x then
        signal missing_from(low)
    end
    while low <= high do
        try: int := (low + high)/2
        if a[try] > x then
            high := try - 1
        elseif x > a[try] then
            low := try + 1
        else
            return(try)
        end
    end
    signal missing_from(low)
end
```

Figure 2

anything to the variables of the actual arguments (like x above). At termination the result objects are assigned to the variables provided.

When a procedure returns one result value, it can be also used as an expression:

if $string\_similar(compar, getl(: input))$**then**
$putl(: output, "Doesnotmatch")$
end

If multiple values are returned, the return values can be assigned to variables using multiple assignment:

quotient, remainder: int := divmod(i, j)

When a procedure executes a **return** statement it is said to terminate normally. Procedures can also terminate exceptionally signalling a name and zero or more signal objects. These exceptional conditions are listed in the procedure header using the **signals** clause. An exceptional termination is caused by execution of a **signal** statement. The signal statement names the exceptional condition and lists the result values.

Often there exist argument values that are not meaningful for a procedure. In these cases the procedure should not return any "default" result or crash, but signal an exception. Procedures that are not prepared for arbitrary argument values, are poor abstractions and of very little use elsewhere than for the exact purpose of their design. Writing procedures properly makes the code reusable. For example, the `binary_search` routine of figure 1 could be written like in figure 2.

The exception names and especially the types of signal values should be selected in a uniform way. The signal names (and types) used in the libraries of XE should also be used by the user in his code in similar situations. For example, if a user-defined procedure is defined to signal `not_possible(int)`, it is very hard to use some built-in operations in the same context, because they have the type `string` connected with the signal name `not_possible`.

If there exists a possibility that the procedure invoked could terminate exceptionally, the signal raised by the invocation should be caught. This is done by an exception handler:

```
position: int := binary_search(table, value)
    except when missing_from(i: int):
        putl(:output, "Not found")
    end
```

When the procedure terminates normally the handler is not executed. Control is transferred to the statement following the handler. When an exception occurs, the corresponding handler branch is executed before the execution continues at the statement following the handler. The signal values are bound to the variables declared by the handler. The variable list declared for signal values (like the *i: int* above) is optional. If no declaration is present, the signal values are discarded.

Sometimes there are quite a number of exceptions that can occur and the same action should be executed for each one of them. These situations can be handled most conveniently in the following way:

```
begin
    command: cmd := read_cmd()
    check(command)
    execute(command)
end
    except otherwise(what: string):
        putl(:output, what)
    end
```

The string type variable (*what*), that is optional, is assigned the name of the signal caught.

Each exception handler catches the exceptions that are raised in the statement it is attached to. By attaching the handler to the end of a **while** loop or to an **if** statement, all exceptions raised by the statements contained are handled. The **begin-end** pair can be used to bind statements together, as in the previous example.

Instead of writing a handler that signals the same signal that it has caught, the **resignal** handler can be used:

```
divmod = proc(x, y: int) returns(int, int) signals(zero_divide)
    quotient: int := x / y
        resignal zero_divide
    remainder: int := x // y
    return(quotient, remaider)
end
```

The reader should notice that exception handlers cannot be attached to expressions (or other constructs) but only to statements. Thus it is impossible, for example, to catch only the exceptions raised by the condition of a **while** loop, but not the ones raised by its body.

Sometimes it is convenient to raise an exception locally, not terminate the procedure. This can be done with the **exit** statement, that transfers control to the corresponding handler

```
for column: int in indexes(:matrix) do
    for row: int in indexes(:matrix[column]) do
        if matrix[column][row] = 0 then
            x := row
            y := column
            exit found
        end
    end
end
    except when found:
    end
```

A **break** statement could not be used in this case, because it would terminate only the innermost loop.

3. Parameterization

All modules, except constants, can be parameterized. This allows a set of related abstractions to be defined by a single module. For example, the `string_similar` example of the previous section is applicable only for strings. However, the same algorithm can be used for any tuple:

```
similar = proc[tuple, element: type](a, b: tuple) returns(bool)
        where tuple has size: proctype(tuple) returns(int),
            indexes: itertype(tuple) yields(int),
            fetch: proctype(tuple, int) returns(element) signals(bounds)
        where element has similar: proctype(element, element) returns(bool)
    if size(:a) ~= size(:b) then
        return(false)
    end
    for i: int in [0 .. size(:a) -1] do
        if ~similar(:a[i], b[i]) then
            return(false)
        end
    end
    return(true)
end
```

Here two *parameters* (*tuple* and *element*) have been inserted into the header. Parameters are declared in the same way as arguments. All constant types and the pseudo-type **type** are allowed to be used as parameter types. This ensures that all types are known at compile time and permits complete compile-time type checking.

A parameterized module must be *instantiated*. That is, we must provide actual, constant values for the parameters. The result of an instantiation is a procedure, iterator, rule, or data type that may be used just like a non-parameterized module of the same kind. For each distinct list of actual parameter values, a distinct procedure, iterator, data type, or rule is obtained.

For example, if `similar` is given types `string` and `char` as parameter values, it generates a procedure that is equivalent to `string_similar`. The instance is made and invoked in the following way:

```
a_is_b: bool := similar[string, char](a, b)
```

The procedure generator `similar` is applicable for a wide range of types. For example, `similar[array[int], int]` and `similar[sequence[string], string]` are legal instances of it.

The parameter values that are acceptable are defined by the restrictions given in the header (the **where** clauses). The restrictions require the parameter types to have operations conforming to a specific interface. The interface is specified by a routine type. In the example, the tuple type is required to have operation `size`: a procedure that has one argument (of the type tuple) and returns an integer.

The routine type can also include signals. The `fetch` routine is *allowed* to signal `bounds` (but it does not have to have the signal specified in its header). The specification **signals(*)** allows a routine to signal anything. In the example it is assumed that the signal `bounds` never occurs. It could be handled by a normal exception handler.

4. Constructing data types in XE

XE is a data abstraction language. The programming style pertinent with XE differs from the traditional top-down approach. Instead of the stepwise refinement method (that is a control-oriented method of programming) XE encourages one to write programs in a data-oriented way. In XE the data abstractions form the building blocks for programs. Before going into detailed plans for control the data abstractions should be designed. This kind of an approach usually leads to significant simplifications in the control, because data is handled abstractly.

An abstract data type is defined by a set of values (or states of objects of that type) and a set of operations. A simple example of an abstract data type is given in figure 3.

Figure 3: Set abstraction for small integer sets.

```
int_set = datatype is new, insert, remove, member

    rep = array[int]

    new = proc() returns(cvt)
        return(rep$new())
    end

    insert = proc(s: cvt, elem: int) returns(cvt)
        if member(up(s), elem) then
            return(s)
        end
        result: rep := rep$alloc(size(:s) + 1)
        i: int := 0
```

```
    while i < size(:s) do
        result [i] := s [i]
        i := i + 1
    end
    result [i] := elem
    return(result)
end
```

```
    remove = proc(s: cvt, elem: int) returns(cvt) signals(not_found)
        for i: int in indexes(:s) do
            if s[i] = elem then
                result: rep := rep$alloc(size(:s) - 1)
                for j: int in [0 .. i - 1] do
                    result[j] := s[j]
                end
                for j: int in [i + 1 .. size(:result)] do
                    result[j - 1] := s[j]
                end
                return(result)
            end
        end
        signal not_found
    end

    member = proc(s: cvt, elem: int) returns(bool)
        for e: int in rep$elements(s) do
            if elem = e then
                return(true)
            end
        end
        return(false)
    end

end
```

The example defines a new data type `int_set` that models sets of integers. It is suitable for handling sets having few elements, because the operations are rather slow for large sets. A new empty set can be created by operation `new`. Operation `insert` constructs a new set which has all the members of *s* and *elem* as its members. Operation `delete` constructs a new set which has all the members of *s*, but not *elem* as its member It signals if *elem* is not a member of *s*. The operations of set can be invoked using the `$` notation:

```
a: int_set := int_set$new()
a := int_set$insert(a, 5)
```

Another way is to use the colon notation:

```
a: int_set := int_set$new()
a := insert(:a, 5)
```

The colon means that the insert operation of the type of *a* is to be invoked.

`Rep` is a special name that defines the representation type. Each set object is represented by an array. The state of an object should be represented by as primitive data types as reasonable (build-in types are the most primitive ones). Even though the representation type is a mutable type, the objects of the type `int_set` are immutable.

The most important thing about abstract data types is that they enforce the hiding of implementations. The user of the data type is able to handle the objects of the type only by the operations provided by the type. So, the user of the `int_set` type can handle its objects only by using operations `new`, `insert`, and `remove`.

The representation operations (above, the operations of array) can be applied only inside the data type definition, where the representation is visible. Inside a data type objects of the type can be handled also abstractly. The operation **up** used in the example converts from the representation type to the abstract type. (There exists also the reverse conversion **down**.) The special type specification **cvt** means that the abstract type is shown outside, but the representation type is used inside.

As for procedures, data types should be written as generic as possible to improve code reusability. Instead of writing a single data type `int_set`, a data type generator `set` could be written:

```
set = datatype[t: type] is new, insert, remove, member
        where t has equal: proctype(t, t) returns(bool)

    rep = array[t]

...

end
```

the body of `set` would be equal to the body of `int_set`, except that parameter *t* would be used instead of type `int` as the element type. The `set` generator can generate a set for each type that has the `equal` operation. For instance, `set[int]` would be equivalent to `int_set`.

4.1 Designing Operations

Data types should be designed following the abstraction they model. All the operations that are part of the abstraction should be included in the data type. If some of the basic operations are left out, the reusability of the code written is usually poor. For example, assume an abstraction for real numbers in a case where only the multiplication and addition operations are needed. Even in this case the subtraction and division operations should be included in the design, because they are a part of the abstraction, even if they are not needed at the moment. If they are left out, the real number abstraction has very little use elsewhere, and cannot be reused in some other program. Adding operations to data types can be very hard later, especially when the operations are in conflict with the selected representation of data.

On the other hand only the operations that are really essential to an abstraction should be included in the corresponding data type. It is impossible (and unnecessary) to predict all the ways in which an abstraction can be used. Assume, for example, that a list abstraction is implemented. An operation for searching the first matching element from a list is needed in this particular program. However, the operation should not be included in the data type, because searching is not a part of the list abstraction, and can be implemented using other operations of the type. If an operation for the search of the first matching operation is included, why not also include the operation for the search of the last matching element? Or, the operation for searching the first different element, or the last matching sublist, or the greatest element, and so on!

However there are some operations that should always be included in a data abstraction if possible: `equal`, `similar`, and `copy`, `parse`, `unparse`, `dump`, and `undump`. If a data type does not have these operations, its use as a component of an data structure is limited. For example, to copy a data structure *all* components are required to have the operation `copy`. Thus a single field in a vast structure can prevent copying the structure. The operations should be defined as for the built-in types.

The inclusion of these "standard" operations is often very simple, because they can be inherited directly from the representation type. For example, the header of the previous example could be rewritten in the following way to define some of the "standard" operations in a proper way.

set = **datatype**[t: **type**] **is** new, insert, remove, member
        inherits equal, similar, copy
    where t **has** equal: **proctype**(t, t) **returns**(bool)

  rep = array[t]

...

end

(Note that the `parse` operation cannot be inherited because it would allow a set to contain a particular element more than once.)

Data types must always have at least one constructor (e.g., the `new` operation of set). Otherwise objects of the type cannot exist, because there is no way to create any. The names for the constructors should be selected following the conventions adopted in the built-in library. The operation name `create` has a special importance (see 4.2), and should be used when possible. Often it is convenient for a data type to also have some conversions to other types. Data types of mutable objects must have at least one operation that can change the state of the objects. It is important to notice that immutable objects can have a mutable representation. (The absence of mutating operations makes the type immutable.) The things connected to mutability are discussed more thoroughly in the XE Reference Manual.

4.2 Syntactic Sugar

XE has several mechanisms to abbreviate statements and expressions. These mechanisms are called *syntactic sugar*, because they do not increase the power of the language; they just make XE programs easier to write and understand. The syntactic sugar also introduces some polymorphic features into the language. The syntactic sugar is also applicable to user defined types.

Literal overloading makes it possible to use literals and constructors with user-defined type. For example, the sequence constructor can be used to create sets, if the `create` operation is included in the data type:

```
create = proc(s: sequence[t]) returns(set[t])
    result: set[t] := set[t]$new()
    for elem: t in elements(:s) do
        result := insert(result, elem)
    end
    return(result)
end
```

Now, it is possible to to represent sets by sequences. The compiler uses the `set[int]$create` procedure to create a set of integers from a sequence of integers:

```
s: set[int] := [1, 2, 4]
```

Operator overloading provides a means of writing expressions in infix notation using mathematical symbols as operators. For example, if the addition and multiplication operations of a user-defined real number type are

```
add = proc(x, y: cvt) returns(cvt) signals(overflow)
    ...
end
```

```
mul = proc(x, y: cvt) returns(cvt) signals(overflow)
    ...
end
```

then arithmetic expressions can be written in the usual form also for the real numbers:

```
r: real := a*(b + c)
```

XE also has sugared statements for selecting and updating components of container types. They can be used instead of the sometimes rather complicated canonical invocations. For example,

```
a[i] := a[j]
```

is actually equivalent to

```
store(:a, i, fetch(:a, j))
```

Therefore selection and update operations of container types should be named `fetch` and `store` (like those of `array`).

XE has a number of other operation names that are connected to syntactic sugar. These names and the required operation types (interfaces) are listed in the XE Reference Manual. The operation name having a sugared form should not be used for purposes that are in conflict with the sugar.

5. Iterators

Iterators are a generalization of the iteration methods available in most programming languages. They permit users to iterate over arbitrary types of data in a convenient way.

An iterator is given zero or more argument objects and it computes a sequence of *items*, that it yields one at a time. Each item is a group of zero or more objects. For example, the following operation yields all the elements of a set:

```
elements = iter(s: cvt) yields(t)
    i: int := 0
    while i < size(:s) do
        yield(s[i])
        i := i + 1
    end
end
```

An iterator can be invoked only by special language constructs: the **for** statement and the **iterate** statement. An invocation of the previous set operation could look like:

```
for e: int in set[int]$elements(x) do
    putl(:output, unparse(:e))
end
```

The **for** statement is executed as follows. The iterator is invoked, and control is transferred to the body of the iterator. If the set is empty the **while** loop is not entered and the iterator is terminated when its end is reached. (It is also possible to terminate the iterator explicitly by a **return** statement with no result values.) No values are yielded and the **for** loop is not entered.

If the set is not empty the **yield** statement is reached. The execution of the iterator is suspended and the result values (the first element of the set) is bound to the variable *e* declared in the heading of the **for** loop. Control is transferred to the body of the **for** loop. After the body has been executed once, it is suspended. (The possible assingments to local variables declared in the body remain in effect.) Execution of the iterator is resumed at the statement following the **yield** statement. This is repeated until all elements of the set are yielded.

It is important to notice that the iteration concept is abstract. The objects can be yielded in any order (or from any structure). The way in which the iteration is actually performed is hidden from the invoker. Thus, it is possible to write, for instance, an iterator iterating over a binary tree:

```
nodes = iter(bt: binarytree) yields(node_type)
    for node: node_type in nodes(bt.left) do
        yield(node)
    end
    yield(bt.root)
    for node: node_type in nodes(bt.right) do
        yield(node)
    end
end
```

The tree is visited in depth first order.

The **for** statement is not able to cope with situations where there is a need for several simultaneous iterations that are not nested within each other. An example of this kind of a situation is the merging of two ordered lists into a single ordered list. The **iterate** statement executes a set of iterators in parallel. The merging example could be solved as follows:

```
merge = proc[t: type](a1, a2: array[t]) returns(array[t])
        where t has gt: proctype(t, t) returns(bool)
    result: array[t] := array[t]$alloc(size(:a1) + size(:a2))
    iterate i1: e1: t in elements(:a1)
            i2: e2: t in elements(:a2) do
        for i: int in [0 .. size(:result) - 1] do
            if ~terminated i2 cand (terminated i1 cor e1 > e2) then
                result[i] := e2
                next i2
            else
                result[i] := e1
                next i1
            end
        end
    end
    return(result)
end
```

The iterations are given unique names (*it1, it2*). Each iterator execution is independent of the others and consists of an iterator to be invoked and a set of variables to which the results yielded are assigned.

The **iterate** statement is not a loop – the body is executed only once. Before the body is entered all iterators are invoked and the objects of the first item returned by each iterator (if any) are assigned to the corresponding variables. Then all the nonterminated iterator executions are suspended and the body is executed.

Inside the body of an **iterate** statement the iterations are controlled by the **next** statement. **next** resumes execution of a iterator to yield the next item (set of objects), assigns them to the corresponding variables, and suspends it again. If the iterator is already terminated and **next** is executed then the values of the corresponding variables become undefined. Reference to any undefined variable of an **iterate** statement causes a **failure** exception to be raised.

The **terminated** expression is used to study the state of execution of an iterator. Its value is **true** if the corresponding iterator has terminated, i.e., the values of the corresponding variables are undefined.

Iterators resemble procedures in many ways. The same statements can be used inside iterators as inside procedures. Iterators can terminate exceptionally like procedures. Signals are declared, raised, and caught as for procedures. Iterator types can be defined in a similar way as procedure types and they can be used in **where** clauses.

6. Rules

A rule is given zero or more argument objects and it creates a sequence of items, that it yields one at a time. Thus the interface of a rule is similar to the interface of an iterator. The items yielded usually contain data that satisfies the *condition* of the rule.

The following is a rather artificial example of rules. A storage room is filled with packages. A package consists of cells. The packages that are not marked or have at least one broken cell are useless, and can be discarded. Also, if all cells of a package have been used and are currently empty the package can be discarded.

```
ruleset = [rule(r: room) yields(int, proctype())
             when [for p: package in packages(:r)]
                    not p.marked or [some c: cell in cells(:p)] broken(:c):
               yield(size(:p),
                     proc() binds(p)
                       discard(p)
                     end)
           end,

           rule(r: room) yields(int, proctype())
             when [for p: package in packages(:r)]
                    [all c: cell in cells(:p)] (used(:c) and empty(:c)):
               yield(size(:p),
                     proc() binds(p)
                       discard(p)
                     end)
           end]
```

Both cases are handled by a rule. The rules are collected together in a ruleset – here a sequence of nameless rules defined as a constant. The **for** quantifier is used to iterate over the packages in the room. The **all** (**some**) quantifier forms a clause that is true if the quantified subclause is true for all values (at least one value). The routine invoked by a quantifier can be a rule or an iterator. The most primitive clauses are predicates (invocations) returning a boolean value, like `p.marked` or `broken(:c)`. Note that special **and**, **or**, and **not** operators are used in clauses instead of the usual boolean operations (see 7.2).

Each time the clause of a **for** quantifier is true, the action part of the rule is executed. The action part consists of a sequence of statements. In the example the actions are single statements that yield a nameless procedure to remove a useless package and the size of the package. The reader should note that a procedure can be treated as an object and it can bind (using the **binds** clause) variables from its lexical environment. The bindings remain in effect even if the routine is executed elsewhere.

The ruleset of the example can be used to remove the biggest useless package from the storage:

```
discard_package = proc(storage: room) signals(no_useless_packages)
    best_action: proctype()
    best_size: int := -1
    for r: ruletype(room) yields(int, proctype()) in elements(:ruleset) do
        for size: int, action: proctype() in r(storage) do
            if size > best_size then
                best_size := size
                best_action := action
            end
        end
    end
    if best_size < 0 then
        signal no_useless_packages
    end
    best_action()
end
```

The rules are applied to the storage room. They yield separate actions for discarding each useless package. The largest package is selected and discarded. As can be seen from the example, also rules can be handled as objects and assigned to variables. Rule types are defined in a similar way as for iterators and procedures.

7. Using Libraries

Libraries have a very important role in XE. The libraries actually form a part of the language. Writing own library modules and reusing code stored in libraries is typical for XE programming. Using the data-oriented design and programming style the libraries tend to grow quite large and rich. Therefore all XE libraries should be consulted before writing a new module. It is quite possible, that the abstraction you need is already implemented.

Especially the built-in library is important for the language and the XE programming style. In the following the types and type generators included in the built-in library are overviewed.

7.1 The Type NULL

The type `null` is meant to be used as a place holder in oneof and variant types. The type has only one object `nil`, so, variables of type `null` are of very little use. The type `null` typically used for representing an undefined value, like:

input_symbol = oneof{character: char, integer: int, error: null}

or, to terminate recursive structures:

```
binary_tree = datatype[data: type] is ...

    rep = variant{empty: null, value: node}

    node = record{contents: data,
                  left, right: binary_tree}

    ...

end
```

7.2 The Type BOOL

The type `bool` has two values **true** and **false**, which represent the logical truth values. The type `bool` has the usual `and`, `or` and `not` operations, which are sugared by `&`, `|`, and `~`, respectively. However, instead of the `and` and `or` operations of the type special language constructs **cand** and **cor** are often used. The **c** in their names stands for conditional evaluation: the second argument is evaluated only if the result cannot be determined from the first. They make conditions easier to write. For example,

```
if size(:s) > 0 then
    if s[0] = 0 then
        ...
    end
end
```

can be written as

```
if size(:s) > 0 cand s[0] = 0 then
    ...
end
```

7.3 The Integer Types

There are three different integer types in the built-in library. The normal signed integer type is `int`. The types `byte` and `word` represent unsigned integers. All integer types are immutable. Therefore constructs like

```
i: int := read()
if state = "update" then
    update_value(i)
end
```

cannot be used in XE (results must always be returned). Because of efficiency, the integer arithmetics of XE are modular, i.e., results of all operations are forced into the allowed range by adding n∗(maxvalue - minvalue + 1) to the original result (where n is an integer). This makes writing of generic modules sometimes very tricky, because it is hard to find out overflows caused by multiplication.

7.4 Strings and Characters

Characters include alphabetic ones (both upper and lower case), digits, and typically some punctuation marks and special symbols. Characters are written in single quotes:

```
c: char := 'a'
```

The actual character set can vary from one implementation to another. Therefore the conversions between character and integer types should be used with care, especially when generic modules are written.

Strings are immutable n-tuples of characters. They are used for representing text, like:

```
s: string := "this is a string"
```

Strings are lexicographically ordered (based on the ordering of characters). String are immutable objects, because this makes manipulation of text much safer. This immutability can sometimes yield inefficiency. For example, long strings should not be constructed from single characters by concatenation, e.g.:

```
result: string := ""
for c: char in elements(:table) do
    result := result || to_string(:c)
end
```

but using arrays:

```
temp: array[char] := array[char]$alloc(size(:table))
i: int := 0
for c: char in elements(:table) do
    temp[i] := c
    i := i + 1
end
result: string := to_string(:temp)
```

7.5 Arrays and Sequences

Arrays are one-dimensional mutable containers. Each array consists of a tuple of elements that are indexed sequentially, starting from 0. All elements of an array are of the same type, and the indexes are of type int.

Array is a type generator. Therefore the element type must be provided in a type specification of form array[*type_spec*] when an array is declared. Examples of array declarations are:

```
array[char]
array[array[string]]
```

Elements of an array can be of any type, mutable or immutable, of static or dynamic size, including arrays themselves. Thus it is possible to create multidimensional objects, as in the second example. Note that size is not a parameter of an array variable, thus all arrays of compatible type are assignment compatible.

Array can be created in many ways. The `fill` operation takes two arguments, the size of the array to be created and an object to fill it with. If the elements of the array are mutable objects, the user must be aware of the fact that `fill` produces an array were all elements actually share the same (given) object. The operation `fill_copy` should be used when sharing is not wanted.

Arrays with uninitialized elements can be created using the operation `alloc` (as in the string construction example above). Care should be taken when the operation alloc is used. Fetching an uninitialized array element returns in a typical implementation something that is not a proper XE object.

Sequences are immutable arrays. Once a sequence is created, its elements cannot be replaced. This makes sequences safer to manipulate than arrays. Especially generic routines should use sequences as their arguments, if side-effects are not needed. (Though, it should be noticed that sequences can contain mutable elements.)

7.6 Tags

The built-in library contains an enumeration type generator `tag`. Tags are typically used as names or flags, like:

weekday = tag{monday, tuesday, wednesday, thursday, friday, saturday, sunday}

A tag literal begins with a backquote:

day: weekday := ‘monday

Days are enumerated starting from 0 by operation `to_int`, and the inverse operation is `from_int`. For example, the following would result in *wednesday*.

the_day_after_tomorrow: weekday := weekday$from_int((to_int(:day) + 2)//7)

7.7 Oneof and Variants

Variants and oneof represent labeled objects. A typical oneof is:

token = oneof{word, reserved: string, integer: int, punctuator: char}

Oneofs are immutable. For example, a token (as defined above) once created, cannot be changed. Variants are mutable. Both the label and value can be changed. An example of a typical variant type is given in Section 7.1.

There exists a special language construct, **tagcase**, for handling objects of types like variant and oneof:

```
t: token := get_next(input)
tagcase t of
  when word(s: string):
      add_symbol(table, s)
  when integer(i: int):
      check_range(i)
  when punctuator(c: char):
      if c = "." then
         add_symbol(table, "get")
      end
end
```

A **tagcase** statement resembles a **case** statement. The expression in the **tagcase** heading is executed. It is compared with the tags in the alternatives and the body of the matching one is executed. As in a **case** statement, there can be an **otherwise** option that is executed if no match is found.

The backquote notation can be used to create oneofs and variants, for example:

t: token := ‘word(”name”).

7.8 Records and Structs

A record is a collection of labeled components. For example:

token = record{contents: string, line: int, column: string}

A record is created using a literal. All fields specified must be assigned a value. The order has no significance:

t: token = {line: 5, contents: ”name”, column: 12}

Components of a record can be selected using a special .-notation:

line: int := t.line

A similar notation exists for updating components:

t.column := 5

Structs are like records, but they are immutable. Therefore fields of structs cannot be updated.

7.9 The I/O

For I/O there is a `stream` type that includes the primitive input actions for reading and writing. There are three streams that are always open: the primary input, the primary output, and the error output. They usually refer to the user's terminal or some console.

The way, in which I/O is performed is implementation dependent, but the following example is quite typical:

```
copy_file = proc()
    pi: stream := stream$primary_input()
    po: stream := stream$primary_output()
    err: stream := stream$error_output()
    puts(:po, "Give source file name: ")
    source: stream := stream$open(getl(:pi), 'read)
        except when not_possible(why: string):
            putl(:err, "Cannot open - " || why)
            return
        end
    puts(:po, "Give target file name: ")
    target: stream := stream$open(getl(:pi), 'write)
        except when not_possible(why: string):
            close(:source)
            putl(:err, "Cannot open - " || why)
            return
        end
    while true do
        putc(:target, getc(:source))
    end
        except when end_of_file:
        end
    close(:source)
    close(:target)
end
```

The example copies contents of a file to another one. File names are asked from the user (using the primary I/O device). The operation `puts` does not output a new line character after is argument string as `putl` does. If errors occur when opening the files, an error message is given and the action aborted. Characters from the source are copied until the exception *end_of_file* occurs.