Foreword

Programming embedded systems is more difficult than ordinary programming and developers of such systems should be provided with at least as good abstraction and expression mechanisms as other programmers. The high verification requirements of embedded systems has lead to the choice of data abstraction as the foundation of XE, and XE supports this methodology in a uniform and general way.

To facilitate the programming of embedded expert systems, XE adds a production rule mechanism to the toolkit of data abstraction languages. In this way it helps the programmer in verifying the correctness of rule based modules and also in enhancing their efficiency by a judicious choice of data structures.

XE owes much to Clu, which has been used as a starting point in its design. XE's mechanisms for defining procedures, iterators and data types will be familiar to anybody knowing Clu. Therefore this manual owes also much to the documents where Clu is described (Liskov et al. 1981).

Programs written in XE are to cooperate with programs and subroutines written in lower level languages, such as PL/M or C. Thus, an interface to these languages has been included in XE, as are some data types required in implementing data structures that are close to the machine level.

The requirement that an XE program can be compiled into efficient code for a 16 bit microprocessor, such as, the Intel 8086 has been a special challenge and has affected every facet of the XE implementation. Hopefully, though, the programmer will not notice these implementation decisions. Thus, we believe that they have not damaged the abstraction level and external appearance of the language proper.

This manual is not a guide to the language – it is only a language reference manual. To learn about the XE language and XE programming style you should read *Programming in XE*. Notational conventions used in this manual are described in Appendix C.

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.

Vesa Hirvisalo is responsible for the final form of this manual. Nassim Bouteldja and Timo Lehtimäki of Telenokia have contributed in an important way by being the first outside users of XE.

1. Introduction

XE is a data abstraction language. It supports a data-oriented programming style. Therefore the concept of user definable data types is fundamental to XE. This section contains a general view of the language and its semantic primitives. Basic concepts and language constructs are described more thoroughly in the other sections of this manual.

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.

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. Procedures can terminate exceptionally signalling a name and zero or more signal objects.

An *iterator* implements iterator abstraction by creating a sequence of *items*, that it yields one item at a time. An item is a group of zero or more objects. As procedures, iterators are given zero or more argument objects. They can terminate in a normal way returning nothing, or exceptionally in the same way as procedures. A typical iterator iterates over a data structure, i.e., yields all elements of the structure one after another. An iterator can be invoked only in special language constructs.

Rules are used to implement a condition-action abstraction. A rule is a special kind of 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.

The modules in an XE program can have a nested structure relative to one another. Thus it is possible to nest one procedure definition inside another to obtain an internal procedure. A nested module cannot directly use variables which are declared outside the module.

The following is a typical procedure definition:

```
gcd = proc(i, j: int) returns(int) signals(undefined)
  % the greatest common divisor is computed by Euclid's algorithm
  if i = 0 cor j = 0 then
    signal undefined
  end
  while i ~= 0 do
    i, j := j // i, i        % multiple assignment; // is the modulo operator
  end
  return(abs(:j))            % the absolute value of j
end
```

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; the number, order, and types of the results returned from the call; and the possible exceptional conditions. In the example,

proc(i, j: int) **returns**(int) **signals**(undefined)

is the header of the procedure *gcd*, which takes two integers as arguments and returns one integer as a result, or terminates exceptionally signaling *undefined.*

The *body* of a procedure definition consists of a sequence of statements and can also contain declarations of local variables. The statements in the body can refer only to these local variables, (nested) modules and the formal names declared in the header.

Within a module, XE uses conventional control structures; e.g., it has the usual constructs like **if then** and **while do**. Comments are introduced by the symbol `%` and continue from there to the end of the line.

1.2 Objects and Identifiers

XE programs refer to *objects* and perform computations on them. Results of operations depend on the *values* of objects. For example, the value of an integer object might be 3.

Each object belongs to a data type. The data type completely determines what kinds of values an object may have and what operations can be performed on it. For example, an `add` operation is meaningful for integer objects but not for string objects.

To operate on objects, programs need some way to refer to them, and to do this they use *identifiers*. The identifiers that may refer to different objects as time goes by are called *variables*. A variable always has a type. The type checking of XE guarantees that a variable always refers to an object of an allowed type. Identifiers are *not* objects, they cannot be referred to and they cannot contain anything (in theory); they are just labels for naming objects.

Objects come into existence though execution of various operations and exist within an *object universe*. For example, the `create` operation for arrays causes a new array object to be created in the object universe. Objects cannot be explicitly removed from the universe so that they can be thought of as existing forever. However, in practice, the space used by inaccessible objects is reclaimed by *garbage collection*. An object is *accessible* if it is referred to by an identifier or another accessible object.

Each object is distinct from all others. Usually an object cannot physically contain other objects, but it can contain references to other objects; when this happens, we say that the containing object *refers to* the object contained. Because 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. Also, it is possible for several objects to refer to the same object. When this happens, we say that they *share* the object referred to.

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. Objects whose value is not guaranteed to be constant are called *mutable*. The current value of the object depends on its *state*. For instance, arrays are mutable: when we change the first element of an array of integers to be a different integer object, the state of that array has changed. Immutable objects are unique; there exists only one object for each value in the universe, but there can be several mutable objects with the same state.

Whether an object is mutable or not depends on the operations that its type supports. If there exist operations that can modify the objects, then the objects are mutable. The mutability of an object strongly affects its abstract behavior and therefore the remarks on mutability in Section 8.4 should be read with care.

When two or more variables (or objects) share a mutable object, changes to the shared object are visible through each of the sharing objects. However, 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. Thus, integers belonging to the value of an object can be physically stored within the object.

1.3 Assignment and Invocation

There are two fundamental actions that relate to objects – assignment and invocation. Assigning an object to a variable gives the object the name of the variable. After an assignment, the variable refers to the object assigned to it. A variable can refer only to one object at a time; the old reference (if any) disappears when an object is assigned to the variable.

In an assignment the object is not copied; an object can have several names, i.e., variables sharing it. Assignment does not affect the value of the object assigned; the object itself remains untouched.

Assignment in XE is of the basic form

variable := expression

where the right hand side (the value of the *expression*) is assigned to the left hand side (the *variable*). In an assignment the type of the expression must be assignment compatible (see Section 8.2) with the type of the variable. Besides this simple assignment XE uses a form of multiple assignment:

variable_1, variable_2, ... := expression_1, expression_2, ...

This is equal to a sequence of simple assignments where all the values of the expressions are first assigned to temporary variables and then the values of these variables are assigned to the variables on the left hand side. This allows the user, for example, to swap the values of two variables without explicit use of temporaries:

a, b := b, a

Instead of a list of expressions the right hand side can consist of a single invocation that returns multiple values, whose number, order and types match the left hand side. There are some forms of *syntactic sugar* that can be used in the left hand side. They are discussed in Sections 1.5 and 4.2.

Besides this kind of explicit assignment XE has a number of constructs performing *implicit* assignment. They are the argument passing mechanism, that is discussed below, and the **signal**, **exit**, **return**, and **yield** statements, that are described in Sections 4 and 7. All assignments can operate on multiple values that are given either by a list of expressions or by a single invocation.

The other fundamental action called invocation has the form

primary([expression, ...])

(Note that we often give simplified syntax definitions to make the semantics obvious. The complete syntax definitions can be found in Appendix D.)

Usually the primary is just the name of a routine, and the expressions are variable names as in the invocation:

parse(input, output, errors)

The sequence of activities in performing an invocation are:

1. The *primary* is evaluated and the result must be a routine, i.e., an iterator, a procedure, or a rule.
2. The expressions are evaluated.
3. New variables corresponding to the formal arguments are introduced.
4. The values of the expressions (actual arguments) are assigned to the corresponding formal arguments.
5. Control is transferred to the routine.

When the invocation is terminated, control returns to the invoker and the result objects are available. If the invocation terminates exceptionally control will not go to the point of return; it is given to an exception handler, as described in Section 7.

1.4 Types

The concept of a data type is fundamental to XE. Each data type in XE is a data abstraction. Thus a data type defines a set of objects and a set of primitive operations to create, examine and manipulate those objects. XE provides a set of built-in types (such as `bool`), built-in type generators (such as `record`), and a mechanism that allows the user to define new types and type generators.

A data type is implemented by a *type module* that describes a *concrete representation* for objects of that data type and routines which implement its operations. The concrete representation is defined in terms of other data types. The data types of XE are called abstract because the concrete representation of an object can be seen only by routines inside the definition of the data type. Objects can be manipulated only via operations of their type; thus the operations of the type, and only they, define the abstract behavior of an object.

In XE it is possible to write *type generators*, i.e., modules that can implement an infinite number of data types. This can be done by parameterization; therefore type generators are also called parameterized types. A type generator is quite similar to an ordinary type, except that the concrete representation and the operations can depend on parameters of the generator. A type generator defines an ordinary data type each time it is provided a list of actual parameters matching those declared in the header of the type generator.

One goal of XE is to make user defined types powerful. Built-in and user defined types are treated as uniformly as possible. Just as built-in types, user defined types can be parameterized and their operations can be freely declared to have parameters, arguments, return values, restrictions and exceptional terminating states. User defined types can also benefit from the *syntactic sugar* of XE, i.e., abbreviations. Only the following restrictions must be observed when defining data types.

1. Built-in types and type generators cannot be redefined or renamed.
2. Some built-in types are related to control constructs of XE. For example type `bool` is used in if-statements as the condition value. There exists no means to use user defined types instead of the built-in ones in such constructs.
3. Only objects of immutable built-in types can be used as constants. There exists no way to use objects of user defined types as constants.

The built-in types of XE are `null`, `bool`, `byte`, `int`, `word`, `char`, and `string`. Type `null` has only one object **nil**, that is used as a place holder in container objects. Type `bool` has two objects **true** and **false** that represent the logical truth values. Type `byte` models bit vectors of typically 8 bits. Bytes can also be treated as a subrange of unsigned integers. Type `word` resembles byte, it typically holds 16 bits or 32 bits. Type `int` provides the usual integer arithmetics. Type `char` models characters and `string` sequences of characters.

The built-in type generators are `array`, `sequence`, `record`, `struct`, `tag`, `variant`, and `oneof`. `Arrays` are static one-dimensional n-tuples indexed by integers. (Arrays of arrays are used for multidimensional arrays.) An array has a high bound and its low bound is always 0; all integers between these bounds (inclusive) are legal indexes into that array. Arrays have operations to create new arrays, fetch and store elements, iterate over elements, etc. All elements of an array must be of the same type, and this type determines the type of the array. For example, `array[int]` is the type of an array containing integer elements, while `array[array[int]]` is the type of an array containing arrays of integers as elements. Note that the array bounds are not part of the type.

Sequences are the immutable form of arrays. Many of the operations of arrays and sequences are similar. A sequence type is generated by the specification `sequence[`*element_type*`]`.

Records, structs, tags, variants and oneofs are examples of type generators definable by *selector type generators*. A selector generator is provided a *list of fields*. A *field* is a pair that consists of a name and a type.

Record objects are mutable and model heterogeneous sets of tagged objects. A record type is generated by a specification providing the fields of the record, for example:

record{length, weight: int, name: string}

Records have operations for creating and copying objects, and updating and selecting their components. Components of a record are identified by names (like *length* etc. above). Structs are immutable records.

Besides the use of names in record definitions, it is possible to create explicit name objects, tags. Tag objects are immutable. A tag type is an enumerated type, it is generated by a specification like:

tag{odd, even: null}

Variant objects are mutable and model tags and tagged objects of heterogeneous types. For example the specification

variant{some: array[int], none: null}

specifies a type, whose objects can refer to an array with tag *some* or consist of only the tag *none* with no value. Oneofs are immutable variants.

Procedures, iterators, and rules have a dualistic nature – besides being executable, they can also be treated as objects (assigned to variables or data structures). Each of them belongs to a specific type (see Sections 6 and 9). For each combination of argument, result, and signal types there exists a distinct procedure type, iterator type, and rule type.

1.5 Syntactic Sugar

XE has several mechanisms to abbreviate statements and expressions. We call these mechanisms *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 applicable to user defined types.

Literal overloading makes it possible to use literals and constructors of some specific type as generic literals. Thus, both assignments

```
a: int := 5
b: byte := 5
```

are legal, even though the values represent different objects of different types.

Operator overloading provides a means of writing expressions in infix notation using mathematical symbols as operators:

```
i: int := 2 * 3 + 5
```

instead of invoking the corresponding operations by name:

```
i: int := int$add(int$mul(2, 3), 5)
```

Thus, each operator corresponds to an operation with a corresponding name, whose data type depends on the operand type.

There exists a short hand form for operation names. If an invoked routine is an operation of an argument type, a colon (:) is placed before the argument to denote the type that the operation belongs to. The example expression above can thus also be written

```
i: int := add(:mul(:2, 3), 5)
```

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, for an array *a* and a variable *x* the following expressions are equivalent:

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

```
temp1 := fetch(:a, j)
temp2 := x
temp3 := fetch(:a, i)
x := temp1
store(:a, i, temp2)
store(:a, j, temp3)
```

There exists syntactic sugar for iterators `T$from_to` and `T$from_to_by`. Instead of the canonical invocations abbreviations can be used. For example, the following **for**-loops are equivalent:

```
for i: int in [1..6] do
...
end
```

```
for i: int in int$from_to(1, 6) do
...
end
```

2. Symbols and Declarations

An XE module consists of a sequence of *tokens* (symbols). Tokens can be words, operators, literals, or punctuation marks. 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.

Punctuation marks are used to group and separate symbols. They are listed in Appendix D.

2.1 Names and Reserved Words

A *word* is a sequence of letters, digits and underscores that begins with a letter or an underscore. Some of the words are reserved, because they are used in constructions of the language. Reserved words are words like **begin**, **end**, and **if**; they are listed in Appendix D.

Words that are not reserved can be used as identifiers. Identifiers are used to refer to objects and language constructs. XE is a language that uses static scoping and different kinds of identifiers (e.g., variable names vs. signal names) belonging to different name spaces. What is referred to by an identifier depends on the construct containing the identifier and on the context where the construct is used. The construct selects the name space, and the context selects the scope where to look for the definition of the identifier (scopes are discussed in section 2.2).

Most constructs in XE use the basic name space that contains variables, arguments, parameters, and constants, but there exist separate name spaces for signals, handlers, iterations, fields, and operation names. Signal names are declared by **signals** clauses and used by **signal** and **resignal** statements. Handler names are declared by **except** statements and used by **exit** statements (implicitly also by invocations). Iteration names are declared by **iterate** statements and used by **next** statements and **terminated** expressions. Each field list declares a set of field names, and each type (type generator) a set of operation names.

Because of the separate name spaces many annoying name conflicts are avoided. For example, using an identifier as a field name of some record does not make it impossible to use the identifier as a variable name or field name of another record. But to avoid ambiguities in expressions a single name space is occupied by variable, argument, and constant names. Therefore in a given context there cannot be, for instance, an argument and a variable with the same name.

There are two ways of expressing that a routine is an operation of a data type. The canonical form (without parameterization) is

type_spec$operation_name stack$pop

(Note that we will often display together a syntactic construct and an example of its use without introducing the example in any way.)

This denotes the operation named *operation_name* of the type named *type_spec*. If the routine is invoked and has an argument of type *type_spec* then a short hand form can be used. The *type_spec*$ can be replaced with a colon that preceeds an argument of type *type_spec*. For example

putl(:output, "Hello")

denotes the operation named *putl* of the type of the argument *output* (the one preceded by a colon). Only one argument can be preceded by a colon (:). If the colon is not used the type of the operation must be given using the compound operation name.

2.2 Scopes

The scope of an identifier can be either local to a scoping unit within a module or nonlocal (external to a given module). The scoping structure of a module corresponds to the nesting structure of its statements and submodules. A scoping unit is a *body* together with an associated "heading". The scoping units are

- from a **proc**, **iter**, **datatype**, or **rule** to the matching **end**
- from a **for**,**while**, or **begin** to the matching **end**
- from an **if** or **elseif** to the next **elseif** of the same **if** statement or to the matching **else** or **end**
- from a **when** to the next **when** of the same construction or to the matching **otherwise** or **end**
- from an **otherwise** or **else** to the matching **end** of the same construction
- from a **use** to the end of the type specification
- from the beginning brace { of a *type set specification* to its ending brace }

New identifiers (*idn* for short) for constants and variables may be introduced in a scoping unit. The scope rules are as follows:

- An idn may not be redefined in its scope, excluding the submodules contained.
- Any idn that is used as an external reference in a scope may not be used for any other purpose in that scope.

Thus, unlike many "block-structured" languages, XE prohibits the redefinition of an identifier in a nested scope (excluding submodules). By ruling out redefinition, XE avoids the "hole in scope" problem that can lead to program errors.

The scope of a variable is from its declaration to the end of the smallest scope unit containing its declaration excluding contained submodules that do not bind it. If a variable is declared with initialization, the initialization expression is not included in the scope of the variable. The scope of a formal argument is the body of the routine, excluding contained submodules that do not bind it. The scope of a constant is the smallest scope unit containing its declaration, including the submodules contained. The scope of a formal parameter is the whole module, including the submodules contained and the header of the module.

2.3 Variable Declarations

Variables are introduced by declarations. Variables must be declared before use. There are two sorts of declarations: with and without initialization. Declarations without initialization take the form

```
idn, ... : type_expr                    x,y: int
```

A declaration without initialization introduces a list of variables, all having the type given by *type_expr*. Such a declaration determines the types of objects that can be assigned to each of these variables. The variables introduced by this kind of declaration are *uninitialized*, i.e., initially they denote no objects.

A declaration with initialization combines declarations and assignments and forms a statement. It is equivalent to one or more simple declarations followed by corresponding assignment statements. A declaration with initialization is either single:

```
decl := expression                      x: int := 3
```

or multiple:

```
decl, ... := expression, ...
```

x: int, b: bool := some_procedure()

In the second form there must be n expressions resulting in n objects of correct types or an invocation giving such results. Variable declarations are allowed wherever statements are allowed.

2.4 Constant Declarations

A *constant definition* (an equate) allows an identifier to be used to denote a value of a *constant expression*. Constant declarations are allowed only at the beginning of a scoping unit. Typically they are used to name common constants, such as `radix = 10`, types generated by type generators, like `ai = array[int]`, or routines like `sort = proc(a: array[int]) ...`. The term "constant" is used in a very narrow sense above: a constant must be computable at compile time.

A *constant expression* is a special language construct or an expression that can be evaluated at compile time to produce an object of a *constant type*. Constants and actual parameter values must be constant expressions. Specifically, the following are constant expressions:

- literals of a constant type
- type sets
- field lists
- constant identifiers
- formal parameters
- invocations of operations of constant types, whose return value is of a constant type, and all operands are constant expressions. Any invocation in a constant expression must terminate normally.

Note that type sets can be constants, but there are no such objects and there is no "type" for type sets. The *constant types* are `bool`, `byte`, `int`, `word`, `char`, `string`, procedure types, iterator types, rule types, the type **type**, and types generated from them by `sequence`, `tag`, `oneof`, and `struct` type generators.

When literals are used in a constant expression the default type of the literal is used if its type is not explicitly given (as explained in Section 2.5).

The syntax of constant definitions is

idn = expression	x = 5 ai = array[int]

Where the expression must be a constant expression. A constant may be used in an expression like a variable, but it is not a variable; it cannot be used on the left-hand side of an assignment statement.

Cyclic dependencies among constant definitions are not allowed. For example,

z = x + y/2	z = x + y/2	
x = 3	x = 3 + z	% Cyclic !
y = x + 2	y = x + 2	

The sequence on the left is a legal sequence of constant definitions, but the sequence on the right is illegal.

2.5 Literals and Constructors

Some built-in types of XE have literals and some type generators have constructors for creating constant objects. Types having literals are `bool`, `int`, `char`, `string`, and all procedure, iterator, and rule types. Type generators `sequence`, `tag`, and `struct` have constructors. The other built-in and user defined types and type generators can benefit from these literals by using syntactic sugar called *literal overloading.*

There are two literals of type `bool`: **true** and **false**, and they represent the logical truth values. Literal overloading of boolean literals is not possible.

Integer literals represent the values of type `int`. By default integer literals are decimal (like 5478), but when necessary any base in the range 2..36 can be used (the corresponding digits are 0..9, A..Z or a..z) by the notation *number*#*decimal_base*, e.g.,:

0FE#16

is equal to 254. (A leading zero must be supplied if the first character would be non-numeric.)

The range of type word contains values not representable as ints. The above notation can, however, still be used. Note that integer literals are always positive, but negative numbers can be written in the normal way using the minus operation of each type.

Character literals represent the values of type `char`. The normal form of a character literal is '*c*', where *c* can be any character except newline. Special characters can be created using the form *ddd*, where *ddd* is some three digit octal number that is in allowed range of characters (see Appendix A.6). Leading zeros can be omitted ('\1' is a legal character). Some control characters are often needed and there exist abbreviations for them. \n means a new line character, and \t means a horizontal tab, \f formfeed, and \b backspace.

String literals represent the values of type `string`. The form of a string literal is "$c_1c_2...c_n$", where each character c_i can be any character except " and newline. Also the special forms of characters above can be used in string literals. Literal "" corresponds to the empty string and the character " can be included in a string by notation \" and \ by \\.

Procedure, iterator, and rule literals represent values of the corresponding types. Each object created by a procedure, iterator, or rule literal is distinct. For instance, values of any two procedure literals cannot be equal. The form of these literals is described in Sections 6 and 9.

Objects of a type `sequence[T]` can be created by sequence constructor literals. Such a literal is of the form $[e_1, ..., e_n]$, where each e_i is an object of type `T`.

Objects of a type `struct`$\{N_1 : T_1, ..., N_k : T_k\}$ can be created by struct constructor. Such a literal is of the form $\{n_1 : e_1, ..., n_k : e_k\}$, where each n_i is one of the names N_i, and e_i an object of the corresponding type T_i. All names N_i must be included in the literal exactly once.

Objects of a type `tag`$\{N_1 : T_1, ..., N_k : T_k\}$ can be created by a tag literal. Such a literal is of the form 'N_i (the character used is a backquote). For example '`even`, represents an object of type `tag{odd, even: null}`.

Objects of a type `oneof`$\{N_1 : T_1, ..., N_k : T_k\}$ can be created by a oneof literal. Such a literal is of the form '$N_i(e)$, where e is an expression of the type T_i. For example '`parsed`(147), represents an object of type `oneof{parsed: int, unparsed: string}`.

Only if all types included in a constructor are immutable built-in types, then the constructor forms a literal that can be used as a constant object.

Literals can also be used by types that have no literals of their own. When a literal is not of the type being assigned to, literal overloading can be used if the type has an operation corresponding to the literal. The operations corresponding to each literal and constructor form are:

$[e_1, ..., e_k]$	T\$create:	**proctype** (sequence[E]) **returns**(T)
$\{n_1 : e_1, ..., n_k : e_k\}$	T\$create:	**proctype** (struct$\{N_1 : T_1, ..., N_k : T_k\}$) **returns**(T)
'N_i	T\$create_N_i:	**proctype**(null) **returns**(T)
'$N_i(e)$	T\$create_N_i:	**proctype**(T_i) **returns**(T)
int literal	T\$create:	**proctype**(sequence[byte]) **returns**(T)
char literal	T\$create:	**proctype**(char) **returns**(T)
string literal	T\$create:	**proctype**(string) **returns**(T)

where T is the type being assigned to, E the element type, and $N_i : T_i$ a field name/type pair. The `sequence[byte]` for an integer literal contains the integer written in the $2^{bytesize}$-base system instead of the normal 10-base system. (So, it contains the minimum number of bytes to hold bits of the corresponding unsigned integer, right justified, and the most significant byte at position 0.) $2^{bytesize}$ is usually $2^8 = 256$. For instance, integer literal 1234 ($= 4 * 256 + 210$) is equivalent to the invocation:

```
create(sequence[byte]$[4, 210])
```

Thus the following assignments can be written even if types `byte`, `word` and `array[int]` have no literals:

```
b: byte := 124
w: word := 0FFFF#16
a: array[int] := [1, 7, 3, 9]
```

Because of literal overloading each literal can denote objects of different types. If necessary, the type of a literal can be given using the form *type*\$*literal* (the *compound literal*), like

```
int$137
array[int]$[6, 3, 2]
```

If the type of a literal is not explicitly given, then it is resolved as follows. If the literal is assigned to a variable or given as an argument (or is the right hand side of any implicit assignment), then it has the same type as the variable or the formal argument, (or the target of the implicit assignment). If the literal is an argument of any infix operator, then the literal has the same type as the other argument. If the type cannot be resolved using the rules above, then the default type of the literal is used (`int` for integer literal, `string` for string literal etc.). For example the following are equivalent:

a = word$1	a = word$1
b = a + 1	b = a + word$1
c: byte := 1 + 2	c: byte := byte$1 + byte$2
d = 1 + 2	d = int$1 + int$2
if a > 0 **then**	**if** a > word$0 **then**
part(:"abcd", i, j)	string$part(string$"abcd", i, j)
end	**end**

Type deduction of infix and prefix operators (+, -, ...) is explained in Section 3.6.

3. Expressions

An expression evaluates to an XE object that is called the *result* or the *value* of the expression. Simple expressions, such as literals, variables, parameters, and routine names directly name their result object. More complex expressions are generally built up by using nested invocations of procedures. The value returned by the outermost invocation is the result of such an expression.

XE has prefix and infix operators for the common arithmetic and comparison operations and allows indexing and component selection using the normal notation:

```
a[i]
r.s
```

All these operation forms are syntactic sugar for canonical invocations. There are only five operations that correspond to no canonical invocation: the conditional operators **cand** and **cor**, the representation type conversion operations **up** and **down**, and the termination predicate **terminated** (see Section 4.6).

Expressions are also used to generate types from type generators. This kind of expressions are described in Section 8.

3.1 Literals

All literals can be used as expressions. The type of a literal is deduced by resolving the literal overloading as explained in Section 2.6.

3.2 Identifiers

Variables are names for objects of a given type. The type of a variable, given in the variable declaration, determines which objects that variable may denote.

Formal arguments are identifiers like variables, but they are declared in the routine header and are used to pass argument objects from the invoker to the invoked routine.

Formal parameters are identifiers for constants supplied when a parameterized module is instantiated. The type of a parameter is given in the corresponding declaration. Parameters of type **type** can be used only in constant expressions (to generate types from type generators).

An identifier defined by a constant definition has the type of the object referred to. If that constant is a type or a type set specification then the identifier cannot be used as an expression.

3.3 Procedures, Iterators and Rules

Any procedure, iterator, or rule can be used as an expression. The value of such an expression is the procedure, iterator, or rule object. The type of such an expression is the type of the routine object as explained in Section 6. When a procedure, iterator or rule is defined as an operation of a data type, the name of the data type is part of the name of the routine (compound names and their syntactic sugar are explained in Section 2.1).

3.4 Procedure Invocations

Procedure invocations can be used as expressions and have the form

primary([expression, ...])	add(2,3)

The invocation is performed as described in Section 1.3. The primary must evaluate to a procedure that returns a single result object. Each expression must be assignment compatible (see Section 8.2) with the corresponding formal argument of the procedure. The type of the entire invocation is the type of the result object. Instead of a list of expressions a single invocation can form the actual arguments if it returns the same number, order, and types of result objects (see Section 1.3).

Procedures can also be invoked as statements. The special case of invoking a selective procedure inside a selector generator is described in Section 5.4.

3.5 Special Operations

Selection operations provide access to the individual elements or components of a collection. A simple notation is provided for invoking the *fetch* operations of types like sequence types, and the *get* operations of types like structure types.

An element selection expression (indexing) has the form

primary[expression]	arr[3]

Let T be the type of primary. Then the above is a short form for the following invocation of a `fetch` operation:

T$fetch(primary, expression)	array[int]$fetch(arr, 3)

The expression is legal whenever the corresponding invocation is legal.

The component selection expression for records and structs has the form

primary.name	David.age

This is a short from for an invocation of a `get_`*name* operation for the type T of primary, and is equivalent to

T$get_name(primary)	person_type$get_age(David)

The expression is legal whenever the corresponding invocation is legal.

3.6 Prefix and Infix Operators

XE allows prefix and infix notation as an abbreviation for several operations. The procedures invoked are listed below.

$expr_1$ `^` $expr_2$	T$power:	**proctype**(T ,T) **returns**(T)
$expr_1$ // $expr_2$	T$mod:	**proctype**(T ,T) **returns**(T)
$expr_1$ / $expr_2$	T$div:	**proctype**(T ,T) **returns**(T)
$expr_1$ `*` $expr_2$	T$mul:	**proctype**(T ,T) **returns**(T)
$expr_1$ + $expr_2$	T$add:	**proctype**(T ,T) **returns**(T)
$expr_1$ – $expr_2$	T$sub:	**proctype**(T ,T) **returns**(T)
$expr_1$ `||` $expr_2$	T$concat:	**proctype**(T ,T) **returns**(T)
$expr_1$ `<` $expr_2$	T$lt:	**proctype**(T ,T) **returns**(bool)
$expr_1$ `<=` $expr_2$	T$le:	**proctype**(T ,T) **returns**(bool)
$expr_1$ `=` $expr_2$	T$equal:	**proctype**(T ,T) **returns**(bool)
$expr_1$ `>=` $expr_2$	T$ge:	**proctype**(T ,T) **returns**(bool)
$expr_1$ `>` $expr_2$	T$gt:	**proctype**(T ,T) **returns**(bool)
$expr_1$ `&` $expr_2$	T$bitand:	**proctype**(T ,T) **returns**(T)
$expr_1$ `|` $expr_2$	T$bitor:	**proctype**(T ,T) **returns**(T)
`~` *expr*	T$bitnot:	**proctype**(T) **returns**(T)
- expr	T$minus:	**proctype**(T) **returns**(T)
$expr_1$ `~=` $expr_2$	T$equal:	**proctype**(T ,T) **returns**(bool)

(The `~=` operator invokes procedure `bool$bitnot` to complement the result of `T$equal`).

For infix operators `^`, `//`, `/`, `*`, `+`, `-`, `||`, `&`, and `|` and prefix operators `~` and `-` the type of the operation is the type of the arguments and of the variable (or target of an implicit assignment), where result is assigned. For infix operators `<`, `<=`, `=`, `>=`, `>`, or `~=` the type of the operation is the type of the arguments. The infix and prefix operations are allowed to signal normally.

3.7 Cand and Cor

Type `bool` is strongly bound into the language. For operating on boolean expressions there exist two special operations **cand** and **cor**, that are not operations of type `bool`, but constructs of the language. The **c** in their names stands for conditional evaluation. The expression

$expr_1$ **cand** $expr_2$

is the boolean and of $expr_1$ and $expr_2$. However, if $expr_1$ is false, $expr_2$ is never evaluated. The expression

$expr_1$ **cor** $expr_2$

is the boolean or of $expr_1$ and $expr_2$, but $expr_2$ is not evaluated unless $expr_1$ is false. For both **cand** and **cor** , $expr_1$ and $expr_2$ must have type bool.

3.8 Precedence and Associativity

Precedence and associativity rules are used to resolve ambiguities of expressions that are not completely parenthesized. The precedence of each operator is given below.

8	`^`, `-` (unary), `~`
7	`*`, `/`, `//`
6	`+`, `-`, `||`
5	`<`, `<=`, `=`, `>=`, `>`
4	`&`, **cand**
3	`|`, **cor**
2	**not**, [**some** ...], [**all** ...]
1	**and**
0	**or**

The associativity for operators of the same precedence is from left to right, except for ^, which is from right to left.

3.9 Up and Down

There exists two special conversions:

> **up**(expression)
> **down**(expression)

They may be used only within the body of an operation of a data type. **Up** changes the type of the expression from the representation type of the objects to the abstract type. **Down** converts the type of the expression from the abstract type to the representation type.

4. Statements

XE statements are executed for their side effects and do not return any values. Thus, XE can be called a statement oriented language. *Control statements* permit the programmer to define the control flow of the program. However, the real work is done by the assignment and invocation statements.

The syntax of XE permits a control statement to control a group of constant definitions, declarations, and statements rather than just a single statement. Such a group is called a *body*. Reserved words belonging to the various compound statements define the end of a body. Occasionally (mainly for purposes of exception handling) we would like a group of statements in straight-line code to be treated like a single statement; this is achieved by enclosing the group of statements between the keywords **begin** and **end**.

Some statements are not described in this section, because they are connected to other structures of the language. The assignment statement is explained in Section 1.3. The statements dealing with exceptions (**signal**, **begin-end**) are described in Section 7.

4.1 Invocation Statement

The form of an invocation statement is the same as that of an invocation expression:

primary([expression, ...])　　　　stream$putl(output, "Hello")

The primary must evaluate to a procedure object. The type of each argument expression must be assignment compatible (see Section 8.2) with the corresponding formal argument. Any results returned by the invocation are discarded. Instead of a list of expressions a single invocation can form the actual arguments if it returns the same number, order, and types of result objects (see Section 1.3).

4.2 Update Statements

Updating components of records and arrays can be performed by two special statements. They resemble assignments syntactically, but are really invocations (see Sections 1.3 and 1.5).

The *element update* statement used with arrays has the form primary[$expr_1$] := $expr_2$ arr[i] := j

This is a shorthand for invoking a `store` operation. It is equivalent to the invocation statement

T$store(primary, $expr_1$, $expr_2$) array[int]$store(arr, i, j)

The *component update* statement used with records has the form

primary.name := expression George.age := 30

This is shorthand for invoking a `set_`*name* operation. It is equivalent to the invocation statement

T$set_name(primary, expression) person_type$set_age(George, 30)

These forms are usable also in multiple assignment statements. For example,

a[i], a[j] := a[j], a[i]

can be used to swap two array elements.

4.3 Conditional Statement

The conditional statement is of the form

if expression **then**
  body
{ **elseif** expression **then**
  body }
[**else**
  body]
end

The expressions must be of type bool and are evaluated in succession until one is found to be true. (Thus, the **elseif** allows one to write a multipath branch.) 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** clause is executed (if an **else** clause exists).

4.4 While Statement

The **while** statement is a loop statement of the form:

```
while expression do
body
end
```

```
while i < n do
i := i + 1
end
```

The body is executed repeatedly as long as the expression, which must be of type bool, remains true. Before each execution of the body the expression is evaluated. Execution of the **while** statement terminates when the expression is false.

4.5 For Statement

An iterator (described in Section 6.2) can be invoked by using the loop statement **for**. Such an invocation produces a sequence of *items* (groups of zero or more objects) one item at a time. The body of the **for** statement is executed for each item.

The **for** statement has the form

```
for [ decl, ... ] in invocation do
body
end
```

```
for i: int in from_to(1, 10) do
j := j * i
end
```

The invocation must correspond to an iterator. The *loop variables* declared are local to the **for** statement. The type, order, and number of loop variables must correspond the items yielded by the iterator.

The **for** statement is executed as follows. The iterator is invoked, and it either yields an item or terminates. In the first case the execution of the iterator is temporarily suspended, the objects in the item are assigned to the loop variables, and the body of the **for** statement is executed. The next cycle of the loop is started by resuming execution of the iterator from where it was suspended. When the iterator terminates, the entire **for** statement terminates. If the **for** statement terminates, this also terminates the iterator.

4.6 Iterate Statement

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. It is of the form

> **iterate** iteration, ... **do**
>   body
> **end**

where

> iteration ::= idn: [decl, ...] **in** invocation

All iterator executions are given a unique name (*idn*). Each iterator execution is independent of the others and consists of an iterator to be invoked and a set of variables to which he results yielded are assigned. The identifiers declared are local to the **iterate** statement and read-only (they cannot be assigned to). The number and type of the identifiers declared must correspond to the iterator invoked. If the iterator yields something and no variables are declared then the results yielded are discarded.

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 can be controlled by the **next** statement.

> **next** idn, ...

resumes execution of the iterator *idn* 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. Note that if an iterator terminates without yielding any items the values of the corresponding variables are undefined when execution of the body begins.

There is a special expression of the form

> **terminated** idn

that can be used to study the state of execution of an iterator. The type of a **terminated** expression is `bool` and its value is **true** if and only if the corresponding iterator has terminated, i.e., the values of the corresponding variables are undefined.

4.7 Continue Statement

The **continue** statement:

> **continue**

terminates execution of the body of the smallest loop statement in which it appears and starts the next cycle (if any) of that loop. The loop statements of XE are **while** and **for**.

4.8 Break Statement

The **break** statement:

> **break**

terminates execution of the smallest loop statement in which it appears. The **break** statement can be used inside the **while** and **for** statements.

4.9 Case Statement

The **case** statement is a special control statement. It can be used in place of several if-then-else statements. The form of the **case** statement is

> **case** expression **of**
>   case_option
>   { case_option }
>   [**otherwise**:
>   body]
> **end**

where

> case_option ::= **when** expression, ... : body

Example:

```
case i of
  when 1: do1()
  when 2: do2()
  otherwise: do3()
end
```

The *expressions* in the *case_options* must be constant expressions (see Section 2.4) and of the same type as the *expression* of the heading (following word **case**). The same constant value cannot appear more than once. The *expression* of the heading is evaluated. If one of the constants expressions is equal to the value of the expression of the heading, the matching body is executed. If no constant matches, the body of the **otherwise** option is executed (if it exists).

4.10 Tagcase Statement

The **tagcase** statement is quite similar to the **case** statement:

```
tagcase expression of
  tag_option
  { tag_option }
  [ otherwise:
    body ]
end
```

where

tag_option ::= **when** name, ... [(idn: type_expr)] : body

Example:

```
tagcase a_variant of
  when taga (aux: int): do_a(aux)
  when tagb: do_b()
  otherwise: do_c()
end
```

The expression must evaluate to an object whose type has the operation

T$tag_of: **proc**(T) **returns**(tag{$N_1 : T_1, ..., N_k : T_k$})

Each *name* in the tag_options must be one of the N_i and must be distinct from all the others. If a declaration (*idn: type_expr*) exists then the type of the expression must also have the operation

T$get_$N_i$(T) **proc**(T) **returns**(T_i)

where N_i must be equal to *idn* and T_i equal to *type_expr*. The scope of the *idn* is only the body of the option having the declaration.

The operation *tag_of* is invoked and the result returned is compared against the names on the *tag_options*. When a match is found the corresponding body is executed. If a variable is declared then the *get_*N_i operation is invoked and the result is assigned to the variable. If no match is found then the **otherwise** option (if any) is executed.

A special form of **tagcase** statement to be used in selector generators is explained in Section 5.4.

4.11 Return Statement

The **return** statement:

return [(expression, ...)]

terminates the execution of the containing procedure, iterator, or rule. For a **return** statement in a procedure, the type of each expression must be assignment compatible (see Section 8.2) with the corresponding return type of that procedure. The expressions (if any) are evaluated and the objects obtained become the results of the procedure. For a **return** statement occurring in an iterator or rule, no results can be returned.

4.12 Yield Statement

A **yield** statement:

yield [(expression, ...)]

may occur only in the body of an iterator or rule, and has the effect of suspending operation of the iterator (or the rule) and returning control to the invoking statement. The values obtained by evaluating the expressions are passed to the invoking statement where they are assigned to the corresponding list of identifiers. The type of each expression must be assignment compatible with the corresponding yield type of the iterator (or the rule). After the body of the **for** loop has been executed or at execution of a **next** statement in an **iterate** statement, execution of the iterator (or the rule) is resumed at the statement following the **yield** statement in the invoked iterator (or rule).

4.13 Static

By default all objects are in the scope of garbage collection. The keyword **static** in front of a statement means that any new objects created within that statement or routines called directly or indirectly from it, are allocated from static storage and are not touched by ordinary garbage collection. For example

> **static** x: array[int] := [1, 2, 3, 4]

Note that if a block structured statement like **begin-end** or **if** is preceeded by **static**, then all objects created within the whole construction are allocated statically.

5. Common Properties of Modules

An XE program consists of a group of modules. Five kinds of modules are provided: data type, procedure, iterator, rule, and constant. Of these procedures, iterators, and rules are referred to as *routines*.

In XE, modules can be nested. All modules can contain other modules that are called *submodules*. XE allows free nesting structures – the submodules can have their own submodules, and so on. Each submodule is defined (and can be used) only in the smallest containing scope. The modules that are not nested in any other module are called *top level modules*. The top level modules are global, they can be referred to in any module. This kind of reference is called an *external reference*.

An identifier with an externally established meaning can be redefined and used locally within a module. However, within a module, an identifier may be used only in a purely local fashion or only in a purely nonlocal (external) fashion, but no identifier may be used in both ways. For example, a top level procedure can be invoked only in those other modules where the name of the procedure is not declared (i.e., given an other meaning that is local). However, a top level procedure (or any other procedure) can always be invoked in its own body, because its name cannot be redefined there.

A submodule cannot directly use variables or arguments declared in the surrounding module; thus there exist no global variables in XE (see Section 2.2). Variables or arguments of the surrounding module can be used in the nested routines only if they are bound by the **binds** clause (see Section 6.1). The specific parts of modules are described in the following sections; here we describe the common features: parameters, restrictions and own variables.

5.1 Parameters

All modules, except constants, can be parameterized. This allows a set of related abstractions to be defined by a single module. In a module header there is an optional *parms* clause of the form:

```
[ parm, ... ]
```

where

```
parm ::= idn, ... : type_expr
```

If the *parms* clause is given, it indicates that the module is parameterized. For example:

datatype stack[t: **type**, n: int] **is** ...

Parameters are declared like arguments. All constant types and the pseudo-type **type** are allowed to be used as parameter types. The actual values for parameters must be constants. This ensures that all types are known at compile time and permits complete compile-time type checking.

The scope rules permit the parameters to be used throughout a parameterized module including the *parms* clause. Type parameters can be used in any type specification, and all other parameters can be used as expressions. Type parameters are often used in defining the types of arguments and results:

p = **proc**[t: **type**](x: t) **returns**(t)

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 instantiation, a distinct procedure, iterator, data type, or rule is obtained.

A parameterized module is most easily understood in terms of rewriting. At instantiation, the actual parameter values are substituted for the formal parameters throughout the module, and the *parms* clause and **where** clause are deleted. The result is an ordinary (non-parameterized) module. In the case of a data type, some of the operations may have additional parameters; further rewriting will be performed when these operations are encountered.

5.2 Restrictions

Restrictions on permissible actual types to be substituted for a type parameter can be specified in the **where** clause. It lists a set of operations (together with their types) that the actual type is required to have. The **where** clause constrains the parameterized module as well; only operations listed in the **where** clauses can be used within the module body as operations of the type parameter.

The form of the **where** clause is

where ::= **where** idn **has** oper_decl {, oper_decl }
    | **where** idn **in** type_set_spec

where

```
oper_decl     ::= idn, ... : type_expr
typeset_spec  ::= { idn | idn has oper_decl, ... { equate } }
              |   idn
```

Examples:

```
where type_a has size: proctype(t) returns(int)
where type_b in { t | t has size: proctype(t) returns(int) }
```

In both the above forms of restrictions, the initial *idn* must be a type parameter. The **has** form lists the set of required operations directly, by means of *oper_decls*. The *type_expr* in each *oper_decl* must name a routine type. Note that if some of the type's operations are parameterized, particular instantiations of those operations must be given. The **in** form requires that the actual type be a member of a *type_set*, a set of types having the required operations. That type set can also be defined as a separate module referred to by its name. The identifiers in the *type_set* must match, and the notation is read like set notation. For example

```
t | t has f: ...
```

means "the set of all types *t* such that *t* **has** *f* ...". The scope of the identifier is the *type_set*.

The **in** form is useful because an abbreviation can be given for a type_set by means of a constant definition. If it is helpful to introduce some abbreviations in defining the type_set, these are given in the optional constant definitions (equates) within the type_set. The scope of these definitions is the type_set.

Within a parameterized data type, a routine may also have a **where** clause in its header to place further restrictions on the data type parameters. This kind of a restriction is called a *secondary restriction*. A routine defining a secondary restriction is included in the type instance only if the restriction is satisfied. Take for example arrays. Any type is permissible as the type of array elements. However, the *similar* operation for arrays requires that the element type have a *similar* operation. Thus, procedure array[T]$*similar* exists only when T$*similar* exists. Note that a routine need not include in its **where** clause any restrictions already included in the data type's **where** clause.

5.3 Use Declaration

When data types are designed, the number of operations included in them should usually be kept small, so that only the operations really essential to the type are defined. Often type generators or other parameterized modules make restrictions on their parameters that exclude a number of types that would be semantically appropriate. Therefore XE has a mechanism to insert operations into a type at instantiation time. This binding mechanism is of the form:

idn[type_expr { **use** ridn = expr } , ...]

The expr must be a routine, routine name, or routine instance. The type of the routine must match the type of *ridn* defined in restrictions of the parameter type *idn*. If *ridn* is not defined in the restrictions, then the declaration is illegal. Each different binding generates a different instance, and therefore a distinct type. For example

am = array[my_type **use** copy = my_copy]

declares *my_copy* to be used as *my_type*$*copy* inside the implementation of the array instance. Existing operations of a type can be redefined. This does not break the data abstraction, because the objects of the type must still be handled abstractly. Use declarations can also be made with field lists, for instance

```
pointer = record{offset: word use default = word$min_value,
                 base: word use default = word$max_value}
```

5.4 Field List Parameters

With the ordinary parameterization it is quite easy to write, for example, generic containers of homogeneous objects (like `array`), but writing generic containers of heterogeneous objects (like `record`) is impossible. Ordinary parameterization can pass only a fixed number of parameters and, for instance, a record type generator with a fixed number of components is of little use. Therefore XE has a mechanism for passing type parameter lists of varying size.

These type parameter lists of varying size are called *field lists*. Types, procedures, iterators and rules can be parameterized with field lists. A generator parameterized by a field list is called a *selector generator*. A field list is of the form:

```
field_list   ::= { [ field_spec, ... ] }
field_spec   ::= name, ... : type_expr
```

A field list declares a list of names with associated types. All the names must be distinct (upper and lower case are equivalent). An example of an instance of a selector generator is:

```
literal = oneof{int_lit: int_token,
                string_lit: string_token,
                char_list: char_token}
```

Now, suppose that there are several such structures in a program and a special formated output routine is needed for all of them. We could write a routine for each structure, like:

```
print_literal = proc(s: stream, l: literal)
   tagcase l of
      tag int_lit:
         putl(:s, int_token$print(literal$get_int_lit(l)))
      tag string_lit:
         putl(:s, string_token$print(literal$get_string_lit(l)))
      tag char_lit:
         putl(:s, char_token$print(literal$get_char_lit(l)))
   end
end
```

With selector generators we can solve all the cases by writing a single routine generator:

```
print_oneof = proc{ordered FL}(s: stream, l: oneof{FL})
      where foreach {F: T in FL} T has print: proctype(T) returns(string)
   tagcase l of
      foreach {F: T in FL}
         putl(:s, T$print(oneof{FL}$get_@F(l)))
   end
end
```

Selector generators are most easily understood in terms of rewriting. In the header a *formal field list* is declared (FL in the example). At instantiation occurrencies of the formal field list are substituted with the actual field list. The syntactic constructs starting with **foreach** are duplicated for each field. **foreach** declares a *formal field name* and a *formal field type* (F and T in the example). In each duplicate these are substituted with the corresponding name and type in the actual field list. Then the parameter declaration and the **where** clauses are deleted from the header. The result is an ordinary (non-parameterized) module. For example, if *print_oneof* is given the same field list as in *literal* then it generates a procedure similar to *print_literal*.

Selector generators can be *ordered* or *unordered*. Fields are sorted in alphabetical order at instantiation if the selector generator is unordered, otherwise the declaration order is used (i.e. ordered means that the user of the selector generator can define the order of the fields). The form of a formal parameter is:

```
parms    ::= { ordered field_list_name }
           | { unordered field_list_name }
```

For example, the header of a selector type generator could be

datatype my_record {**unordered** FL} **is** create, get_{FL}, set_{FL}

This selector generator is not ordered. The selector type generator `my_record` could be used to generate, for example, the following type:

stack = my_record{contents: array[int], top: int}

The operations `get_` and `set_` are declared to be *selective*, and must be defined by selective constant definitions (equates). A *Selective definition* looks quite similar to an ordinary one, but it is preceded by the selection clause:

foreach { field_name : field_type **in** field_list } idn = constant

The definitions created are identified by concatenated names of the form *p*@*f*, where *p* is the name of the selective definition and *f* the formal field name. If we need, for example, a procedure created by a selector procedure *get_* of selector generator `struct{FL}`, we can refer to it by `struct{FL}$get_@F`.

At instantiation a selective definition *p* creates one constant *pn* for each field from the actual field list, where *n* is the actual field name. For example, the instance `stack` of the selector generator `my_record` has operations `create`, `get_contents`, `get_top`, `set_contents`. If the header of `get_` is

foreach {F: T **in** FL} get_ = **proc**(r: cvt) **returns**(T)

then the procedures created by `get_` would have headers:

get_contents = **proc**(r: cvt) **returns**(array[int])
get_top = **proc**(r: cvt) **returns**(int)

Restrictions on field types can be defined by a selective **where** clause. It is satisfied only if all the types of the actual field list satisfy it.

where foreach {field_name: field_type **in** field_list} field_type ...

In selector generators a special form of the **tagcase** statement can be used. The form is:

tagcase tag_expr **of**
foreach {field_name : field_type **in** field_list}
...
end

The *tag_expr* must have the operation `tag_of` (see 4.10). Each value of `tag{`*field_list*`}` denotes a distinct field in the actual field list. This kind of **tagcase** statement creates an ordinary **tagcase** statement as in the example at the beginning of this section. The field denoted by *tag_expr* is selected and the corresponding body is executed.

5.5 Own Variables

Occasionally it may be desirable to have a module retain information internally between invocations.

Procedures, iterators, data types and rules can all retain information by using own variables. Such a variable is similar to an ordinary variable, except that it exists for the whole life of the program. It is not bound to the life of any particular routine activation. Syntactically, own variable declarations must appear at the beginning of the data type or routine body (among equates); they cannot appear in bodies nested within statements. Own variable declarations have the form

own_var ::= **own** decl, ...
          | **own** decl, ... := expression, ...

Own variables are created at the start of program execution. The own variables of a routine (including data type operations) are initialized in textual order when the routine is invoked for the first time. At the top level of the body of a data type each declaration forms a group variables that is initialized (in their textual order) when some member of the group is used or a routine that binds the variable is invoked.

Except for the above characteristics (placement of declarations, initialization and lifetime) own variables can be used like other variables.

Own variable declarations in different modules always refer to distinct own variables. Own variable declarations within a parameterized module produce distinct own variables for each instantiation of the module.

Own variable declarations cannot be enclosed by an **except** statement. If initialization of the own variables raises a signal, then a **failure** exception is signaled by the routine invocation that has caused the initialization to take place.

With own variables, procedures and iterators become potentially mutable objects. In general, own variables should not be used to modify the abstract behavior of a module.

zap = **proc**()

signals()

6. Routines

In this section we shall discuss only nonparametrized procedures and iterators, without *parms* and *where* clauses. These features common to all modules, have been described in Section 5. Rules are described in Section 9.

6.1 Procedures

A procedure has zero or more *arguments*, and it terminates by returning zero or more *results*. It may terminate in the *normal* condition, or in one of its *exceptional conditions*. It may return different numbers and types of results in the different conditions.

The form of a procedure is

> **proc** args [returns] [signals] [binds]
>   { equate_or_own }
>   { statement }
> **end**

where

args	::=	([decl, ...])
returns	::=	**returns** (type_expr, ...)
signals	::=	**signals** (exception, ...)
binds	::=	**binds** (idn, ...)

Example:

```
is_rprime = proc(a, b: int) returns(bool)
    return(a // b ~= 0 cand b // a ~= 0)
end
```

The procedure header describes how the procedure communicates with its caller. The *args* clause specifies the number, order, and types of arguments and the **returns** clause specifies the number, order, and types of results returned at normal termination, i.e., when a **return** statement is executed or the end of the procedure body is reached. If the **returns** clause is missing no results are returned by the procedure.

The **signals** clause specifies the exceptional conditions in which the procedure can terminate. It specifies their names and the number, order, and types of result objects returned for each exception. Beside the exceptions explicitly named, any procedure can terminate in the **failure** exception which returns one result, a string object. All names of exceptions must be distinct, and none can be **failure**.

At invocation of a procedure the actual arguments are assigned to variables introduced to represent the formal arguments and the procedure body is executed. Execution terminates at a **return** or a **signal** statement or at reaching the textual end of the procedure body. Result objects of an executed **return** statement are passed to the invoker. If a procedure should return results but reaches the textual end of its body, a **failure** exception is signaled.

If a procedure is nested inside another module it can bind the variables and formal arguments declared in the smallest containing module. The identifiers to be bound must be listed in the **binds** clause. Bound variables of the containing module can be used like local variables. The binding is done when the procedure name or literal is used. Variable bindings remain in effect, even if the procedure is treated as an object and used elsewhere than in its own lexical environment. Thus it is possible to use *lexical closures* in XE.

A non-parameterized or an instantiated parameterized procedure is an object of some procedure type. A procedure type specification is of the form

> **proctype**([type_expr, ...]) [returns] [signals]

where

> signals ::= **signals** (exception, ...)
> | **signals** (*)

A procedure belongs to the procedure types that have the same number, order, and types of argument and result values, and at least the signals the procedure has (with the same number, order, and types of signal values). A procedure type specification with **signals**(*) includes all those procedures that have the same number, order, and types of argument and result values, regardless of their signals.

Procedure objects are handled as if they were immutable. All procedure types have operations equal, similar, and copy with interfaces as defined in Section 8.4.

6.2 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 computes a sequence of *items*, one at a time. Each item is a group of zero or more objects. The form of an iterator is

```
iter args [ yields ] [ signals ] [ binds ]
    { equate_or_own }
    { statement }
end
```

where

yields ::= **yields** (type_expr, . . .)

Example:

```
sizes = iter(a: array[string]) yields(int)
    for s: string in elements(:a) do
        yield(size(:s))
    end
end
```

There are only two differences in the form of an iterator and the form of a procedure:

1. The header of an iterator contains a **yields** clause instead of the **returns** clause of a procedure. This clause specifies the number, order, and types of objects passed to the invoker each time the iterator produces the next item in the sequence. If no objects are yielded, the **yields** clause is omitted.
2. Within the iterator body, the **yield** statement is used to transfer the next item in the sequence to the invoker. An iterator terminates in the same way as a procedure, but it cannot return any results.

A non-parameterized or an instantiated parameterized iterator is an object of some iterator type. An iterator type specification is of the form:

itertype([type_expr, . . .]) [yields] [signals]

where

```
signals ::= signals ( exception { , exception } )
          | signals (*)
```

The types an iterator belongs to are defined in the same way as for procedures (see Section 6.1).

Like procedures, iterators can be used to form lexical closures and the iterator types have the operations equal, similar, and copy.

There exists syntactic sugar for some iterators. The abbreviations and the corresponding operations are:

[$expr_1$..$expr_2$]	T\$from_to:	**itertype**(T, T) **yields**(T)
[$expr_1$..$expr_2$; $expr_3$]	T\$from_to_by:	**itertype**(T, T, T) **yields**(T)

This form of syntactic sugar should be used only with iterators that have similar semantics as `int$from_to` and `int$from_to_by` (see Appendix A.4).

6.3 Foreign Procedures

XE programs communicate with other programs by calling procedures written in other languages. Even an operating system is in this sense just another program. Beside communication, such *foreign procedures* can also be used for other purposes.

The form of a foreign procedure declaration is:

idn = **foreign** idn **proc** str args [returns]

The *args* and *returns* are declared as for normal XE procedures. The equated identifier is the internal name of the procedure that is used for invoking it. The *idn* after the reserved word **foreign** is the name of the foreign language and *str* is a string literal defining the external name of the procedure (e.g., its name in a library). Foreign procedures cannot signal. Example:

foreign plm **proc** "rec_msg"(comp: byte, msg: array[byte]) **returns**(word)

The argument passing mechanism for foreign procedures differs from argument passing for XE procedures. The argument and result objects are always copied (foreign procedures are not allowed to use the XE address space). XE objects are made linear when they are copied; this is done by `dump`-operations of the argument types. Result objects are constructed into XE form by `undump`-operations of the result types. Therefore only types having operation `dump` can be used as arguments and types having `undump` as results. All built-in types and type generators have these operations (not listed in Appendix A):

T\$dump = **proc**(b: binary, o: T) **signals**(end_of, not_possible(string))

Description	Writes a copy of the object o of type T to binary b.
Signals	end_of if it is not possible to write more data into the binary; not_possible(s) if an error occurs, s is a description of the error.

T$undump = **proc**(b: binary) **returns**(T) **signals**(end_of, not_possible(string))

Description Reads an object of type T from binary b, and returns a copy of it.

Signals end_of if there are no objects left in b; not_possible(s) if an error occurs, s is a description of the error.

The user can write `dump` and `undump` operations for his own types and thus define the external memory formats (dump formats) for his objects as he wants. The dump formats of XE objects are implementation dependent.

Foreign procedures are invoked in the same way as normal XE procedures. Each foreign procedure is a member of all procedure types having the same number and order of argument and result types.

7. Exception Handling

7.1 Raising Signals

When a routine activation terminates by signaling an exception, the corresponding invocation is said to *raise* that exception. Inside the routine body the exceptional termination can be caused by the **signal** statement, which has the form

signal signal_name [(expression, ...)]

This statement may appear anywhere in a routine body. When it is executed the expressions, if any, are first evaluated to produce a list of *exception results*. The activation of the routine is then terminated and execution continues in the caller.

The *signal_name* signaled must be either one of the exception names listed in the routine header or **failure**. **failure** is a special signal that is meant for system errors, like lack of memory space, and it should not be used for any other purposes.

If the exception has specification (in the **signals** clause)

name(T_1, ..., T_n)

then a **signal** statement raising that exception must contain n expressions with the type of the i^{th} expression being assignment compatible with T_i. If the name is **failure**, one signal object must be given (the type of the object is implementation dependent).

7.2 Exit Statement

Signals can be raised also locally by using an **exit** statement, which has the form

exit signal_name [(expression, ...)]

This statement is similar to a **signal** statement, except that it *raises* the exception directly in the current routine, whereas the **signal** statement *signals* an exception to the *calling* routine. An exception raised by an **exit** statement must be handled explicitly by a handler with the same name and the same types in its formal objects list, defined in a containing **except** statement.

7.3 Handling Signals

The action to be taken when an exception is raised can be specified by attaching *handlers* to statements. A statement with handlers is called an **except** statement and has the form

```
statement
  except
    { when_handler }
    [ others_handler ]
    [ failure_handler ]
  end
```

where

when_handler	::=	**when** name, ... [(decl, ...)] : body
	|	**when** name, ... (*) : body
others_handler	::=	**otherwise** [(idn : type_expr)] : body
failure_handler	::=	**failure** [(idn : type_expr)] : body

Example:

```
a := b/c
  except when zero_divide:
    a := 0
  end
```

Each when_handler specifies one or more exception names and a body. The body is executed if an exception with one of those names is raised by an invocation in the statement to which the handler is attached. All the names listed in the when_handlers must be distinct. The optional others_handler is used to handle all exceptions not explicitly named in the when_handlers. Handlers can be attached to any form of statement, even another **except** statement.

If, during the execution of a statement S, some invocation in S raises an exception E, control immediately transfers to the closest applicable handler – that is, the closest handler for E that is attached to a statement containing the invocation. When execution of the handler is complete, control passes to the statement following the one to which the handler is attached. If execution of S completes without raising an exception, the attached handlers are not executed.

An exception raised inside a handler is treated as any other exception: control passes to the closest handler for that exception. Note that an exception raised in some handler attached to S cannot be handled by any handler attached to S;

either the exception is handled within the handler or it is handled by some handler attached to a statement containing S.

The form

when name, ... [(decl, ...)] : body

is used to handle exceptions when the exception results are of interest. The optional declared variables, local to the handler, are assigned the exception results before the body is executed. Every exception potentially handled by a handler of this form must have the same number of results as there are declared variables, and the type of the results must equal (rather than include) the types of the variables. The form

when name, ... (*) : body

handles all exceptions with the given names, regardless of the existence of exception results; any actual results are discarded. Hence exceptions with differing numbers and types of results can be handled together.

The form

otherwise [(idn : type_expr)] : body

is optional. This form handles any exception not handled by other handlers in the list, except the **failure** signal. If a variable is declared, it must be of type string. The variable, which is local to the handler, is assigned a lower case string representing the actual exception name; any results are discarded. Note that exception results are ignored when matching exceptions to handlers; only the names of exceptions are used.

An invocation need not be surrounded by **except** statements that can handle all potential exceptions. In many cases a particular exception can be proved not to arise. An example is the invocation *div(x,3)*, which will never signal *zero_divide*. There is a possibility that some invocation may raise an exception for which there is no handler. If this happens, a **failure** exception is signaled. **failure** signals can be handled only by a failure handler.

7.4 Passing signals

A **resignal** statement is a syntactically abbreviated form of exception handling:

statement
    resignal signal_name, ...

Each name is distinct and either one of the condition names listed in the routine header or **failure**. The **resignal** statement is like an **except** statement with a handler for each named condition. In lieu of each such handler, it simply signals that exception with exactly the same results. Thus a **resignal** statement naming an exception with the specification

name(T_1, ...,T_n)

behaves exactly as the statement

when name (x_1:T_1, ..., x_n:T_n): **signal** name(x_1, ..., x_n)

As with explicit handlers every exception potentially handled by an implicit one must have the same number of results as declared in the exception specification, and the types of the results must equal the types listed.

7.5 Begin Statement

The **begin** statement groups a sequence of statements together into a single statement. Its form is

begin body **end**

As bodies are already permitted inside control statements the **begin** statement is mainly used to group statements together for use with the **except** statement.

zap = **proc**()

8. Types

8.1 Type Specification

In declarations, constant definitions (equates) and invocations types are denoted by type specifications. Usually, a type is specified by what is called a type expression, but also a type module is a legal type specification. For a type with no parameters the type expression is just the identifier equated to the type. For parameterized types, the type expression consists of the identifier together with the parameter values.

```
expr ::= idn
      |  idn [ param_value, ... ]
      |  idn { [ field_spec, ... ] }
      |  idn { field_list_name }
      |  routine_type
```

Here *idn* names the type or the type generator. If it is a normal type generator the constants of the appropriate type must be supplied in the order specified by the type. If it is a selector generator appropriate fields must be supplied (see Section 5.4).

Procedures, iterators, and rules are XE objects; they can be passed as arguments and returned as results, assigned to variables and stored as components of composite objects. Procedure, iterator, and rule types are described in Sections 6 and 9.

8.2 The XE Type System

XE is designed to allow complete compile-time type checking. The type of each variable is known by the compiler. Furthermore, the type of any object that could result from the evaluation of any expression (invocation) is known at compile time. Hence, every assignment can be checked at compile time to make sure that the variable is only assigned objects of its declared type.

The rule is that an assignment $v := E$ is legal only if the set of objects defined by the type of E is included in the set of all objects that could be denoted by v. Usually this means that the types of the left and right hand sides of an assignment must be equal. But when routine objects are assigned, the type of the left hand side must include the type of the right hand side, i.e., the routine type of the left hand side must have same order, number, and types of arguments and result values, and at least the signals of the routine type of the right hand side.

When we speak about types, distinction must be made between types and type generators. Objects are of some specific type. The type of an object cannot be a type generator, because type generators can be used only to generate new types.

Objects can be converted from a type to an other only by using operations of the types, or by **up**, **down**, or **cvt** (see section 8.3). There exists no other implicit or explicit type conversions in XE.

8.3 Type Definition

A **datatype** module (often called data type for clarity) is used to implement a new data type, distinct from any other built-in or user defined ones. We shall discuss nonparameterized data types without the *parms* and *where* clauses. These clauses can be added as described in Section 5. The form of a type definition is

> **datatype** [**is** idn , ...] [**inherits** idn, ...]
> { equate_or_own }
> **end**

Also types can be parameterized (see Section 5).

Example:

```
char_set = datatype is create, insert, delete, size, member, elements
        inherits equal, similar, copy

    rep = record{contents: array[char],
                 size: int}

    ...
end
```

The operations of the data type are named by the list of *idns* following the reserved words **is** and **inherits**. All of them must be distinct. The *idns* following **is** must name a routine defined in the *type_body* The *idns* following **inherits** must be operations of the representation type (see below).

To define a new data type it is necessary to choose a *representation* for the objects of the type. The special equate rep = type_spec

within the data type body identifies *type_spec* as that representation. Within the data type, rep may be used as an abbreviation for *type_spec*.

The identifier naming the data type can be used in the data type body. This permits the definition of recursive types.

Beside specifying the representation of objects, the data type module must implement the primitive operations (procedures and iterators) of the type. Routines in the body of the data type usually define *public* operations (those whose names are listed in the data type header). Any additional routines are *hidden*. They may not be named directly by users of the abstract type. All the routines must be named by distinct identifiers whose scope becomes the entire data type.

Outside the data type module, objects of the type may only be treated abstractly (i.e., manipulated only by the public operations). To implement the operations, it is, however, usually necessary to manipulate the objects in terms of their representation. Inside the data type it is possible to view the objects of the type either abstractly or concretely. The syntax specifies unambiguously, for each variable referring to one of the objects of the type, which view is meant. Inside a data type named T, a declaration

v: T

indicates that the object referred to by v is treated abstractly. Conversely, a declaration

w: rep

indicates that the object referred to by w is treated concretely. The primitives, **up** and **down**, can be used to convert between these two points of view. With **up** an object of type rep can be treated abstractly, while **down** forces an abstract object to be viewed concretely. Thus, the following two assignments are legal for the declarations just given:

v := **up**(w)
w := **down**(v)

Only routines inside a data type module are allowed to use **up** and **down**. Note that they merely inform the compiler which view is being taken on an object.

The view of an object is apt to change at the interface to one of the type's operations. Outside the operation the object is viewed abstractly, while inside it is viewed concretely. To facilitate this change of viewpoint, there is a special type specification, **cvt**. Its use is restricted to the *args*, **returns**, **yields** , and **signals** clauses of routines inside a data type and it may be used at the top level only. When used inside the *args* clause, **cvt** specifies **down** to be applied implicitly to the argument object passed by the invoking routine. When **cvt** is used in the **returns**, **yield**s, or **signals** clause, it specifies **up** to be applied implicitly to the result object as it is returned. Thus **cvt** means abstract outside, concrete inside.

When deriving the type of each routine from its header each occurrence of **cvt** is replaced by the abstract type.

The operations named by *idns* following the word **inherits** are said to be inherited from the representation type. The operations inherited have the same number, order, and types of arguments and result values, and the same signals with the same number, order, and types of signal values, except that each occurrence of the representation type will be replaced with the abstract type. For example, the type `char_set` (see above) will have the operation

equal: **proctype**(char_set, char_set) **returns**(char_set)

because of the `equal` operation inherited from the representation type (record).

8.4 Mutability

Mutability of objects affects the operations the corresponding data type can have. As explained in Section 1.2 immutable objects are (in theory) unique. For example, all integer literals 5 denote the same abstract object because no distinction can be made between them by operations of type `int`. Mutable objects have a state that can change, and therefore are *not* unique. For instance, two empty arrays created by the `new` operation can be found distinct by performing the `equal` operation.

All XE data types should have operations `equal`, `similar` and `copy`. The data types of mutable objects should also have operations `equal_state` and `copy_state`. The operations should be defined to work as follows:

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

Description For types having mutable objects returns **true** iff both arguments are the same object (are the same physical object). For types having immutable objects returns **true** iff both arguments have the same value (are the same logical object).

similar = **proc**(a, b: T) **returns**(bool)

Description Returns **true** iff the states of the arguments consist of similar objects (as determined by `similar` operations of their types). For types having atomic objects, returns **true** iff contents of the argument objects are equal.

copy = **proc**(a: T) **returns**(T)

Description Creates an object whose state consists of copies of the objects of the state of the argument. For types having atomic objects, returns an object with equal contents.

equal_state = **proc**(a, b: T) **returns**(bool)

Description Returns **true** iff the states of the arguments consist of equal objects.

copy_state = **proc**(a: T) **returns**(T)

Description Creates an object whose state consists of the same objects as the state of the argument.

For some types the operations `equal`, `similar`, and `copy` need the corresponding operation(s) of their component type(s). Therefore they must have the **where** clauses for these operations.

9. Rules

Rules are used to implement a condition-action abstraction. In XE, a rule is a special kind of iterator that computes a sequence of *items*. An item yielded by a rule usually contains data that satisfies the *condition* of a rule and an action that should be applied to the data. It may also contain data that is used when comparing the items.

A non-parameterized rule is of the form:

```
rule            ::= rule args [ yields ] [ signals ] [ binds ]
                        { equate_or_own }
                        when condition :
                        [ statements ]
                    end
```

Each rule has a fixed number of arguments and it can be parameterized just like a procedure or an iterator. The optional *yields* declaration in the header declares the number, order, and types of the items. The body of a rule consists of a *condition*, a list of statements.

A condition is of the form:

```
condition ::= [ for [ decl, ... ] in invocation ] condition
           |  clause
clause    ::= clause and clause
           |  clause or clause
           |  not clause
           |  ( clause )
           |  [ some [ decl, ... ] in invocation ] clause
           |  [ all [ decl, ... ] in invocation ] clause
           |  predicate
predicate ::= expr
```

Every clause has a value which is either **true** or **false**. A predicate is an expression of type `bool`. The value of the predicate is the value of the expression. Complicated clauses can be built by using operators **and**, **or**, and **not**. The value of an **and** clause is **true** iff the value of both operands is **true**. The value of an **or** clause is **false** iff the value of both operands is **false**. The value of a **not** clause is **true** iff the value of the operand is **false**. The precedences of the operators are given in Section 3.8.

Clauses can be quantified with **all**, **some**, and **for** quantifiers. **all** and **some** quantifications have a value, but a **for** quantification does not (see below). A *quantification* consists of a quantifier, a list of variable declarations, and an iterator invocation. The number, order, and types of the variables declared must match with the number, order, and types of the yielded values of the iterator. The declarations may be omitted, in which case any values yielded are discarded. The scope of the variables declared is the quantified clause, except for the scope of the variables declared by a **for** quantification, which is the quantified condition and the expression part.

The value of an **all** clause is evaluated in the following way. The iterator is invoked and the values yielded are assigned to the variables declared (if any). The value of the quantified clause is then evaluated. If it is **false** the evaluation of the **all** clause is terminated with the value **false**. Otherwise the iterator is resumed and the quantified clause is re-evaluated with the new variable bindings. This is continued until the iterator terminates, in which case the value of the **all** clause is **true**, or until the value of the quantified clause is **false**.

The value of a **some** clause is evaluated in the following way. The iterator is invoked and the values yielded are assigned to the variables declared (if any). The value of the quantified clause is then evaluated. If it is **true** the evaluation of the **some** clause is terminated and its value is **true**. Otherwise the iterator is resumed and the quantified clause is re-evaluated with the new variable bindings. This is continued until the iterator terminates, in which case the value of the **some** clause is **false**, or until the value of the quantified clause is **true**.

Rules can be invoked in **for** and **iterate** statements just like iterators. The invocation is performed as follows. The actual argument objects are assigned to the formal arguments of the rule. The iterator of the outermost **for** quantification of the condition (if any) is invoked. If it yields items the objects are assigned to the quantification variables (if any). If it terminates the execution of the rule terminates. Nested **for** quantifications are executed in the same way, except that when the iterator of a nested quantification terminates the iterator of the closest surrounding **for** quantification is resumed. If the innermost **for** quantification (if any) yields an item the clause contained in the condition is evaluated. If its value is **true** the expressions are evaluated and the values are yielded and the rule is temporarily suspended. If the value is **false** the innermost **for** quantification is resumed and the clause is re-evaluated with the new variable bindings. This is continued until the clause is **true** or the iterator of the outermost **for** quantification terminates.

When the rule is resumed the suspended iterator of the innermost **for** quantification is resumed and the execution is continued as explained above until the rule yields a new item or it terminates.

For example,

```
trivial = rule(i, j: int) yields(int, int)
  when i > j :
    yield(i, j)
end
```

yields its arguments (i, j) if $i > j$ and otherwise nothing. The rule

```
simple = rule() yields(int, proctype())
  own po: stream := stream$primary_output()
  when [for i: int in [1 .. 3]]
    [for j: int in [1 .. 3]]
      i > j :
    yield(i - j,
          proc() binds(po, i, j)
            putl(:po, unparse(:i))
            putl(:po, unparse(:j))
          end)
end
```

yields three items. The first contains integer 1 and an action that outputs "2" and "1" to the primary output. The second contains integer 2 and an action that outputs "3" and "1" to the primary output. The third contains integer 1 and an action that outputs "3" and "2" to the primary output.

Note that the rule abstraction does not contain concepts such as rule set, working memory, conflict resolution, and rule firing, which are typically used in rule based programming. These concepts are usually implemented as follows:

rule set	A data structure containing rules and providing an iterator over these rules.
working memory	A data structure of XE objects providing one or more iterators to be used in the conditions and clauses of rules.
conflict resolution	A routine that compares two items and retains the "better" one.
rule firing	An invocation of an action embedded in an item.

Note that the condition part of the rule should not contain invocations having side-effects.

A non-parameterized or an instantiated parameterized rule is an object of some rule type. A rule type specification is of the form:

> **ruletype**([type_expr, ...]) [yields] [signals]

The rule types that a rule belongs to are defined as for procedures and iterators (see Section 6).

Appendix C: Notation

In this manual we have used extented BNF to describe the syntactic structures of the language. A BNF definition consists of productions, each defining one non-terminal symbol.

nonterminal ::= *BNF-expression*

The non-terminal symbol is defined by a BNF expression – a sequence of symbols (terminal or non-terminal) and operators. The notation has the following operators:

{ a }	A sequence of zero or more *a*'s.
a, ...	A sequence of one or more *a*'s separated by commas.
[a]	An optional *a*.
a | b	An *a* or a *b*.

Reserved words appear in bold face. Sometimes the BNF operator characters ({, }, [,], |) are used as terminal symbols. As terminal symbols they appear in different font (`{`, `}`, `[`, `]`, `|`).

Sets are written in the form

$\{s_1, s_2, ..., s_n\}$

where the s_i are the elements of the set s. The order of the elements in a set makes no difference. Tuples are written in the form

$\langle e_1, e_2, ..., e_n\rangle$

where the e_i are the elements of the tuple e in their proper order. If several objects have a common property, say a bound $high$, the object we are talking about is identified by a subscript. Thus, for example, $high_a$ is the high bound of object a.

We have used the normal $\in, \notin$ notation to denote membership.

Appendix D: Syntax

Reserved words of XE are

all	**else**	**inherits**	**proctype**	**true**
and	**elseif**	**is**	**resignal**	**type**
begin	**end**	**iter**	**return**	**unordered**
binds	**except**	**iterate**	**returns**	**up**
break	**exit**	**itertype**	**rule**	**use**
cand	**failure**	**next**	**ruletype**	**when**
case	**false**	**not**	**signal**	**where**
continue	**for**	**of**	**signals**	**while**
cor	**foreach**	**or**	**some**	**yield**
cvt	**foreign**	**ordered**	**static**	**yields**
datatype	**has**	**otherwise**	**tagcase**	
do	**if**	**own**	**terminated**	
down	**in**	**proc**	**then**	

Punctuation symbols and operators are

<table>
<tr><td>”</td><td>string quote</td><td>.</td><td>get, set</td><td>@</td><td>of field</td></tr>
<tr><td>#</td><td>integer base</td><td>..</td><td>from_to (_by)</td><td>[</td><td>param, index, cons</td></tr>
<tr><td>$</td><td>compound symbol</td><td>/</td><td>div</td><td>\</td><td>quote</td></tr>
<tr><td>%</td><td>comment</td><td>//</td><td>mod</td><td>]</td><td>param, index, cons</td></tr>
<tr><td>&</td><td>and</td><td>:</td><td>declaration</td><td>^</td><td>power</td></tr>
<tr><td>’</td><td>char quote</td><td>:=</td><td>assignment</td><td>‘</td><td>tag</td></tr>
<tr><td>(</td><td>grouping</td><td>;</td><td>separator</td><td>{</td><td>fields, cons</td></tr>
<tr><td>)</td><td>grouping</td><td><</td><td>less than</td><td>|</td><td>or</td></tr>
<tr><td>*</td><td>mul, anything</td><td><=</td><td>less or equal</td><td>||</td><td>concat</td></tr>
<tr><td>+</td><td>add</td><td>=</td><td>equal, equate</td><td>}</td><td>fields, cons</td></tr>
<tr><td>,</td><td>separator</td><td>></td><td>greater</td><td>~</td><td>not</td></tr>
<tr><td>-</td><td>minus, sub</td><td>>=</td><td>greater or equal</td><td>~=</td><td>not equal</td></tr>
</table>

Unary and binary operators and their precedence are described in Sections 3.6 and 3.8. Literals and constructors have the forms described in Section 2.5. Syntax of the language is below. Identifiers (*idn*) begin with a letter or underscore and consist of letters, underscores and digits.

args ::= ([decl { , decl }])

binds ::= **binds** (idn { , idn })

<table>
<tr><td>body</td><td>::=</td><td>{ equate }
{ statement }</td></tr>
<tr><td>case_option</td><td>::=</td><td>when expr { , expr } : body</td></tr>
<tr><td>clause</td><td>::=</td><td>clause and clause</td></tr>
<tr><td></td><td>|</td><td>clause or clause</td></tr>
<tr><td></td><td>|</td><td>not clause</td></tr>
<tr><td></td><td>|</td><td>(clause)</td></tr>
<tr><td></td><td>|</td><td>[all [decl { , decl }] in invocation] clause</td></tr>
<tr><td></td><td>|</td><td>[some [decl { , decl }] in invocation] clause</td></tr>
<tr><td></td><td>|</td><td>expr</td></tr>
<tr><td>condition</td><td>::=</td><td>[for [decl { , decl }] in invocation] condition</td></tr>
<tr><td></td><td>|</td><td>clause</td></tr>
<tr><td>decl</td><td>::=</td><td>idn { , idn } : primary</td></tr>
<tr><td>equate</td><td>::=</td><td>[foreach] idn = expr</td></tr>
<tr><td>equate_or_own</td><td>::=</td><td>equate</td></tr>
<tr><td></td><td>|</td><td>own_var</td></tr>
<tr><td>exception</td><td>::=</td><td>name [(primary { , primary })]</td></tr>
<tr><td>exception_handler</td><td>::=</td><td>resignal signal_name { , signal_name }</td></tr>
<tr><td></td><td>|</td><td>except { when_handler }
[others_handler]
[failure_handler]
end</td></tr>
<tr><td>expr</td><td>::=</td><td>primary</td></tr>
<tr><td></td><td>|</td><td>(expr)</td></tr>
<tr><td></td><td>|</td><td>~ expr</td></tr>
<tr><td></td><td>|</td><td>- expr</td></tr>
<tr><td></td><td>|</td><td>expr ^ expr</td></tr>
<tr><td></td><td>|</td><td>expr // expr</td></tr>
<tr><td></td><td>|</td><td>expr / expr</td></tr>
<tr><td></td><td>|</td><td>expr * expr</td></tr>
<tr><td></td><td>|</td><td>expr + expr</td></tr>
<tr><td></td><td>|</td><td>expr - expr</td></tr>
<tr><td></td><td>|</td><td>expr || expr</td></tr>
<tr><td></td><td>|</td><td>expr < expr</td></tr>
</table>

| expr > expr
| expr = expr
| expr ~= expr
| expr >= expr
| expr <= expr
| expr & expr
| expr **cand** expr
| expr | expr
| expr **cor** expr

failure_handler ::= **failure** [(idn : primary)] : body

field ::= name : expr

field_spec ::= name { , name } : param_value

foreign_proc ::= **foreign** idn **proc** string_literal args [returns]

foreach ::= **foreach** { idn : idn **in** idn }

invocation ::= primary ([[:] expr { , [:] expr }])
| [expr .. expr [; expr]]

iterator ::= **iter** [parms] args [yields] [signals] [binds] { restriction }
{ equate_or_own }
{ statement }
end

oper_name ::= idn [{ idn }]

others_handler ::= **otherwise** [(idn : primary)] : body

own ::= **own** decl { , decl } [:= expr { , expr }]

param_value ::= expr { **use** equate }

parms ::= [decl { , decl }]
| { **ordered** idn }
| { **unordered** idn }

primary ::= **true**
| **false**
| [primary $] int_literal
| [primary $] char_literal

```
                |   [ primary $ ] string_literal
                |   [ primary $ ] tag_literal
                |   [ primary $ ] oneof_literal
                |   [ primary $ ] { [ field { , field } ] }
                |   [ primary $ ] [ [ expr { , expr } ] ]
                |   idn
                |   idn @ idn
                |   primary . name
                |   primary [ expr ]
                |   primary [ param_value { , param_value } ]
                |   primary { [ field_spec { , field_spec } ] }
                |   primary { idn }
                |   primary $ idn
                |   primary $ idn @ idn
                |   invocation
                |   up ( expr )
                |   down ( expr )
                |   terminated idn
                |   procedure
                |   iterator
                |   rule
                |   foreign_proc
                |   proctype ( [ primary { , primary } ] ) [ returns ] [ signals ]
                |   itertype ( [ primary { , primary } ] ) [ yields ] [ signals ]
                |   ruletype ( [ primary { , primary } ] ) [ yields ] [ signals ]
                |   cvt
                |   type_def
                |   type
                |   { idn | idn has decl { , decl } { equate } }
                |   { [ field_spec { , field_spec } ] }

procedure       ::= proc [ parms ] args [ returns ] [ signals ] [ binds ] { restriction }
                    { equate_or_own }
                    { statement }
                    end

ref             ::= idn
                |   primary . name
                |   primary [ expr ]

restriction     ::= where [ foreach ] idn has decl { , decl }
                |   where [ foreach ] idn in primary

returns         ::= returns ( primary { , primary } )
```

rule ::= **rule** [parms] args [yields] [signals] [binds] { restriction }
{ equate_or_own }
when condition :
{ statement }
end

signals ::= **signals** (exception { , exception })
| **signals** (*)

signal_name ::= name
| **failure**

statement ::= decl { , decl } [**:=** expr { , expr }]
| ref { , ref } **:=** expr { , expr }
| invocation
| **while** expr **do** body **end**
| **for** [decl { , decl }] **in** invocation **do** body **end**
| **iterate** { idn : [decl { , decl }] **in** invocation } **do** body **end**
| **next** idn { , idn }
| **break**
| **continue**
| **if** expr **then** body
{ **elseif** expr **then** body }
[**else** body]
end
| **case** expr **of**
case_option { case_option }
[**otherwise** : body]
end
| **tagcase** expr **of**
tag_option { tag_option }
[**otherwise** : body]
end
| **return** [(expr { , expr })]
| **yield** [(expr { , expr })]
| **signal** signal_name [(expr { , expr })]
| **exit** signal_name [(expr { , expr })]
| **begin** body **end**
| **tagcase** expr **of**
foreach
body
end
| statement exception_handler

	|	**static** statement
tag_option	::=	**when** name { , name } [(idn : primary)] : body
type_def	::=	**datatype** [parms] [**is** idn { , idn }] [**inherits** idn { , idn }] where { equate_or_own } **end**
when_handler	::=	**when** name { , name } [(decl { , decl })] : body
	|	**when** name { , name } (*) : body
where	::=	{ restriction }
yields	::=	**yields** (primary { , primary })