Obliq

A lightweight language for network objects

Luca Cardelli

Digital SRC

Abstract

Obliq is small, statically-scoped, untyped language. It is object-oriented, higher-order, concurrent, and distributed. State is local to an address space, while computation can migrate over the network. The distributed computation mechanism is based on Modula-3 network objects.

Contents

1. Concepts

1.1 The language

1.1.1 Overview

Obliq is small statically-scoped untyped language that supports distributed object-oriented computation. An Obliq execution may involve multiple threads in an address space, multiple address spaces and machine architectures over a local network, and multiple networks over the Internet.

Obliq programs are untyped; this makes the language simpler to implement and, probably, more fun to use. Untyped computations are easier to distribute over the network. However, Obliq is designed so that it is not incompatible with static typing in any fundamental way. Type-unsafe computations produce clean errors, so the run-time is "strongly typed".

Although untyped, Obliq adopts a strict static scoping policy. Static scoping is a necessary condition for any reasonable programming language; without it, one cannot be sure of the meaning of any program identifier. In a distributed language, stating scoping ensures that computations have a precise meaning even when moving across a network; a meaning that is independent of the location where they are executed.

A main feature of the language is its simple approach to objects. Objects are sets of methods, with three operations: selection, update, and cloning. Objects provide also the main mechanism for distributed execution, mimicking Modula-3's network objects. By the mediation of a name server, a program can invoke methods of objects residing in other address spaces.

State, for example the internal state of an object, is local to an address space and is never copied over the network. Computation, in the form of procedures or methods, can be freely transmitted over the network. Actual computations (not their source texts) are transmitted, along with their statically scoped global variables. Hence, migrating computations may carry with them references to pieces of state residing in various address spaces.

1.1.2 Distributed semantics

The Obliq distributed semantics is based on the notions of *values*, *locations*, and *address spaces*. In the syntax, *constant identifiers* denotes values, while *variable identifiers* denote locations. The value contained in a location may be updated by assignment to the variable denoting the location. Each location is relative to a fixed address space.

A value is either a *basic value* (such as a string or an integer), a *closure* (the result of evaluating a method or a procedure), an *object*, or an *array*. A value may have *embedded locations*. An array value has embedded locations for its elements, which can be updated. An object value has embedded locations for its fields and methods, which can be updated and overridden. A closure value may have embedded locations because of global variables in its program text referring to locations in the surrounding lexical scope.

Obliq values may be *transmitted* (copied) over the network. However, embedded locations remain rooted to the address space where they were allocated. When transmitting a value over the network, embedded locations are automatically converted to network references to location in their original address space(s).

Therefore, when holding an Obliq value one may or may not, transparently, be holding a collection of network references to locations in different address spaces. In particular, when holding a closure value with embedded locations, one may be holding a computation that, when executed, accesses its global variables over the network. A computation can migrate over the network, but static scoping (network-wide!) guarantees that its meaning is not affected by the location where it is executed except possibly by the parameters that are given to it at that location.

Objects and arrays, being collection of locations spanning a single address spaces, are themselves rooted to a fixed address space (in fact, network references are generated to them, not to each of their embedded locations.) The immobility of objects and arrays is not a strong limitation, because objects can be *cloned* to a different address space, and because procedures can be sent around that allocate objects and arrays in different address spaces.

In conclusion, the network transparency of Obliq computations makes it easy to build applications that span several address spaces. By the same token, network transparency makes it hard to understand where things are happening. Even when executions happen in the wrong place (by some measure of "wrong") they behave correctly. Obliq relieves some of the burden of distributing computations, but care must still be taken to achieve good distributed performance.

1.2 The implementation

The current Obliq implementation is based on an abstract-syntax-tree interpreter, which could easily be replaced by a byte-code interpreter (or, less easily, by a native-code compiler). The language does not depend on interpreter technology, except that a native-code compiler would be inadequate in heterogeneous hardware environments. On the other hand, the language does not depend on compiler technology either: it can be implemented as a responsive interactive system.

The current interpreter takes advantage of static scoping to generate procedure and method closures that embed only the global locations required by the procedure or method. This technique reduces traffic when computation migrates over the network.

Obliq is available both as a program and as a collection of Modula-3 interfaces. Via the interfaces, Obliq can be embedded in Modula-3 applications, and can both call and be called from Modula-3. Several embedding options are available; with/without various libraries, and with/without the parser.

Applications that embed Obliq can communicate over arbitrary connected locations on the Internet by exchanging Obliq values. The emphasis here is on exchanging *values*, not *text*: the receiving application need not parse the incoming messages. Since values include computations, applications can exchange *procedure values* (as opposed to *program text*) which do not require parsing. The meaning of a procedure value is independent of its network location, even when global variables are involved. In other words, exchanging procedure values avoids all the context-dependent binding problems that are implicit in exchanging program text over the network, and that result in dynamic scoping.

1.3 The object model

1.3.1 Objects and methods

An Obliq object is a collection of fields containing either values or methods. Each field is identified by a label that is unique within the object. Syntactically, an object has the form:

$\{x_1 \Rightarrow a_1, \ldots, x_n \Rightarrow a_n\}$

where n>=0, and "x_i" are distinct field-name identifiers. Each "a_i" is either a method term, indicating a method field, or any other term, indicating a value field.

A method term has the form:

$\text{meth}(x, x_1, \ldots, x_n)\ a\ \text{end}$

where the first parameter "`x`" is self, and the other parameters, for n>=0, are supplied on method invocation. The body of the method is the term "`a`" , which computes the result of method invocation.

1.3.2 Object operations

These are the operations on objects "`o`", in addition to object creation:

`o.x`	Select a value from a value field, and return it.
`o.x(`a_1`, ... ,`a_n`)`	Invoke a method from a method field, supplying parameters, and return the result.
`o.x:=a`	Update a value field "`x`" of "`o`" with a new value "`a`", or override a method field "`x`" of "`o`" with a new method "`a`", returning a trivial value.
`clone(`a_1`, ... ,`a_n`)`	Return an object whose components are the union of those of a_1`, ... ,`a_n (duplicated field names produce an error). The new object shares the methods and field values of the source objects, but replicates their (immediate) state.

1.3.3 Examples

The object below has three components. (1) A value field "`x`". (2) A method "`inc`" that increments "`x`" through self (the "`s`" parameter), and returns self. (3) A method "`next`" that invokes "`inc`" through self, and returns the "`x`" component of the result.

A "`let`" declaration binds the object to the identifier "`o`":

```
let o =
  { x => 3,
    inc => meth(s,y) s.x := s.x+y; s end,
    next => meth(s) s.inc(1).x end }
```

The following are some of the operations that can be performed on the object "`o`".

```
o.x          Selecting the "x" component, producing 3.
o.x := 0     Resetting the "x" component to zero.
o.inc(1)     Invoking a method, with parameters.
o.next       Invoking a method with no parameters (o.next() is also valid).
o.next := meth(s) clone(s).inc(1).x end
             Overriding "next" so that it no longer changes its host object.
```

1.3.4 Self-inflicted operations

The three basic operations on objects, selection, update, and cloning, can be performed either as *external* operations on an object, or as *self-inflicted* operations through self. The concept of self-inflicted operations is used later in the contexts of object protection and synchronization.

When a method or procedure operates on a separate object, those object operations are said to be external, not involving self. When a method operates directly on its own self we say that the operations are self-inflicted. If "op" is either select, update, or clone, then "op(o)" is self-inflicted iff:

"o" is the same object as the self of the currently executing method (if any).

Equivalently, if we are looking at a particular method of the form:

```
m  =  meth(s,...) ... op(o) ... end
```

then the operation "op" is self-inflicted iff:

(1) "o" is the same object as "s" (i.e. self), and

(2) "op(o)" is not within another method or procedure body inside of "`meth(s,...)...`".

Condition (1) is a run-time condition. Condition (2) is a static condition, but is equivalent to asking that the run-time stack frame for the execution of "op(o)" is the one established by "m". Moreover, since forking a thread involves a procedure call, condition (2) implies that the thread executing "op(o)" is the same one executing "m" (see section 1.6).

An operation is said to be external if it is not self-inflicted in the sense above.

1.4 Local state and distributed computation

1.4.1 State is local

Every entity that contains mutable state in Obliq is, potentially, a network handle. (The term *network handle* is used here for a NetObj.T value [Modula-3 Network Objects --], reserving the term *network object* for Obliq objects.) That is, either the entity and its state are contained in the local address space, or the entity is a surrogate pointing to a remote address space, where the real entity and its state reside.

Access and update of remote state involves network communication, but is otherwise handled transparently in the same manner as access and update of local state.

There are three kinds of entities in Obliq that have state:

objects	every field of an object is mutable
arrays	every element of an array is mutable
variables	every variable is mutable (constant identifiers are not)

The state associated with these entities is never duplicated or transmitted over the network. However, network handles to these entities are free to travel. Any attempt to pass state over the network results in passing a handle; remote operations on the handle are reflected back to the local state.

Entities without state are copied for transmission over the network. Mixed entities are copied up to the point where they refer to entities with state.

1.4.2 Computation is distributed

Operations on objects, arrays, and variables have the following semantics when applied to network handles.

When a value field of a remote object is selected, its value is returned over the network to the local address space (this value could be a network handle). When a field of a remote object is updated, or when a method is overridden, a value or method is sent over the network and installed into the remote object. Selection and update operations on remote variables and arrays work similarly.

When a method of a remote object is invoked, the arguments are sent over the network, the result is computed remotely, and the final value (or error, or exception) is returned to the local address space.

When a collection of remote or local objects are cloned, the clone is created at the local site.

Object creation is always local, although one can transmit a procedure that creates an object at a remote site

It is interesting to compare the invocation of a remote method with the invocation of a procedure stored in the value field of a remote object. In the first case, the computation is remote, as described above. In the second case, the procedure is first transmitted from the remote object to the local address space, by the semantics of field selection, and then executed locally.

1.4.3 Computation migrates

In several situations above we said that a method or a procedure is transmitted over the network. Actually, what is transmitted is a *closure* of the method or procedure.

A closure consists of two parts: (1) the internal representation of the source text of a method or procedure, and (2) the values of all the free identifiers in the static context of evaluation.

Some of the free identifiers of a method or procedure may denote entities with state. Then only the corresponding network handles are transmitted with the closure. When the closure is activated remotely, operations on those handles will be reflected to their respective address spaces.

1.5 Protection

Objects can be declared *protected* by enclosing them in a different style of brackets, as show below. Protected objects allow update and cloning operations only when such operations are self-inflicted (see section 1.3.4).

{| x_1 => a_1, ... ,x_n => a_n |}

Protection is particularly useful to prevent clients from corrupting remote servers.

1.6 Synchronization

Synchronization of concurrent threads of execution is an issue in Obliq because a server can be accessed concurrently by multiple remote clients, and also because local concurrent threads can be created (see section 3.3.8). Hence, access to objects and other entities with state must be regulated.

We say that an object is synchronized when (1) in presence of multiple threads, at most one method of the object can be executing at any given time, but (2) a method may call a sibling (that is, another method of the same object) through self, without deadlocking. It is often, but not always, desirable to use synchronized objects when multiple threads of control are a possibility.

The obvious approach to build synchronized objects, which is adopted by many concurrent languages, is to associate a mutex with the object. The mutex is locked when a method of the object is invoked, and is unlocked when the method returns, guaranteeing condition (1). This way, however, we have a deadlock whenever a method calls a sibling. We find this unacceptable, because it prevents perfectly innocent programming styles.

The natural way to satisfy condition (2) is to use reentrant mutexes. This solution, however, seems too liberal, because it also allows a method to call an arbitrary method of a different object, or an arbitrary procedure, which then can call back a method of the present object without deadlocking. This goes well beyond our simple desire that a method should be allowed to call its siblings.

We solve this dilemma by adopting an intermediate locking strategy, which we call self synchronization, based on the notion of self-inflicted operations described in section 1.3.4.

An simple but expressive synchronization mechanisms is built into Obliq objects. The thread module can be used to implement more complex synchronization schemes, if needed (see appendix C).

Synchronized objects have a hidden associated mutex, called the object mutex. An object mutex serializes the execution of field selection, method execution, update, and cloning operations on its object. Here are the simple rules of acquisition of the hidden object mutexes:

> Self-inflicted operations never acquire the mutex of their object. (Note that a self-inflicted operation can happen only after the activation of an external operation on the same object, executed by the same thread, which has locked the mutex.)
>
> External (non self-inflicted) operations always acquire the mutex of an object.

Therefore, a method can modify the state of its host object and can invoke other methods through self, without deadlocking. A deadlock will still occur if, for example, a method invokes a procedure or a method of a different object that then attempts an operation on the originating object.

The synchronization status of an object is specified by the following declarations, which must follow the left bracket of a normal or protected object, and be followed by a comma:

`sync none`	no synchronization
`sync self`	self synchronization

In absence of these declarations, the defaults are:

> The default synchronization for ordinary objects is no synchronization.
>
> The default synchronization for protected objects is self synchronization.

Object synchronization can by itself solve common mutual exclusion problems. For example, when a synchronized object implements a network server, the object's methods are guaranteed not to run concurrently.

Other situations require conditional synchronization, for example when implementing a queue object accessed by multiple threads. The following statement should be used inside the methods of a synchronized object; hence, it is evaluated with the object mutex locked:

```
watch c until guard end
```

A new condition "`c`" can be created by "`condition()`", and signaled by "`signal(c)`". The "`watch`" statement evaluates the condition, unlocks the object mutex (so that other methods of the object can execute) and waits for the condition to be signaled. When the condition is signaled, the object mutex is locked, and the boolean guard is evaluated. If the guard is false, the object mutex is unlocked again, and the watch resumes for a new signal. If the guard is true, the watch statements terminates returning "`ok`", and leaving the object mutex locked. See section 5.6 for an example.

There is no automatic synchronization for variables or arrays.

2. Implementation

2.1 The top-level

The Obliq program, when executed, enters an interactive evaluation loop. At the prompt, "`-  `", the user can input a *phrase*, which is always terminated by a semicolon ";" . The first phrase to try out is probably:

```
- help;
```

which provides basic on-line help on various aspects of the system. More extensive documentation, including this manual, is universally available on the World Wide Web, Digital SRC Entry Point ("`http://src-www.pa.dec.com/src.home.html`").

The most common kind of input phrase is a *term* phrase, which causes the parsing, evaluation, and printing of the result of an expression. Examples of term phrases (and comments) are:

```
- 3+4;          (* question *)
7               (* answer *)

- "this is" & " a single text";
"this is a single text"

- 3 is 4;       (* test for equality *)
false
```

Definition phrases are used to bind identifiers to values in the top-level scope. One can use "`var`" for binding values to updatable variables, "`let`" for binding values, including procedures, to constant identifiers, and "`let rec`" for defining recursive procedures.

```
- var x = 3;

- x := x+1;

- let y = x+1;

- let rec fact =
    proc(n)
      if n is 0 then 1 else n * fact(n-1) end
    end;
```

The Obliq top-level is statically scoped, just like the rest of the language. Hence, redefining an identifier at the top-level simply hides its previous incarnation and does not affect terms that already refer to it.

When a top-level phrase finishes executing, the interpreter pretty-prints the result up to a small default depth, printing ellipses after that depth. One can require a larger (but finite) print depth by inserting an exclamation mark before the final semicolon of a phrase; for example: "`fact!;`". This larger default depth is sufficient in most situations. Otherwise, a given print depth "`n`" can be forced by saying "`fact!n;`".

Closures are printed by printing their program text only. If there are global variables, these are indicated by "`global(`x_1`,...,`x_n`)`" followed by the program text. To print the values of global variables, say "`help flags;`".

2.2 Program files

Obliq programs should be stored in files with extension "`.obl`". Such files may contain any sequence of top-level phrases. Files can then be loaded into the system, with the same effect as if they were typed in at the top-level.

The top-level phrase:

```
- load Foo;
```

attempts to load the file "`Foo.obl`" along the current search path. Alternatively, one can say "`load "Foo.obl";`" to load the same file (relative to the current search path), or provide an explicit file path within double quotes.

The search path is set by the environment variable OBLIQPATH, and can be changed via the "`sys`" built-in module (see the appendix, or "`help sys;`").

At startup time, the Obliq system looks for a file called "`.obliq`" in the user's HOME directory, and loads if it finds it. You may want to put there the line:

```
process_new(["/proj/mips/bin/netobjd-ip"], true);
```

to make sure that netobjd-ip, the network objects name server, is running. (Netobjd-ip exits if it finds another copy of itself running.)

2.3 Modules

An Obliq source file can be turned into a *module* by inserting a module declaration at the beginning. Module declarations are useful because they record source file dependencies. When loading a module the dependent modules are automatic loaded; duplicated loading is avoided.

In addition, modules help keep the top-level consistent when reloading modules, for example after a bug fix. Basically, reloading a module is like rolling back in time to the point when the module was first loaded: all intervening top-level definitions are discarded. This roll back affects only on the top-level definition environment: it does not undo state changes.

It is recommended that a program file `Foo.obl` start with the line:

```
module Foo;
```

and end with the line:

```
end module;
```

A module named "`Foo`" terminating with "`end module;`" is said to be *closed*: any top-level identifier "`x`" declared within "`Foo`" is accessible as "`Foo_x`" outside of "`Foo`" (the syntax "`m_x`" is used also for the built-in modules; see the appendix). If, however, "`end module;`" is omitted, the module is said to be *open*: its identifiers are accessible simple as "`x`". Closed modules should be the norm, but open modules are useful for importing definitions into the top level, and for "pervasive" often-used definitions.

If the module `Foo` relies on definitions stored in other programs files (which should similarly start with a `module` line), then one can start `Foo` with the line:

```
module Foo import Foo2,Foo3;
```

The effect of "`import`" will depend on whether the imported modules are open or closed.

When issuing the command "`load Foo;`", the module declaration guarantee two properties: (A) if the modules `Foo2` and `Foo3` have not been loaded already, they are loaded before `Foo` is loaded;

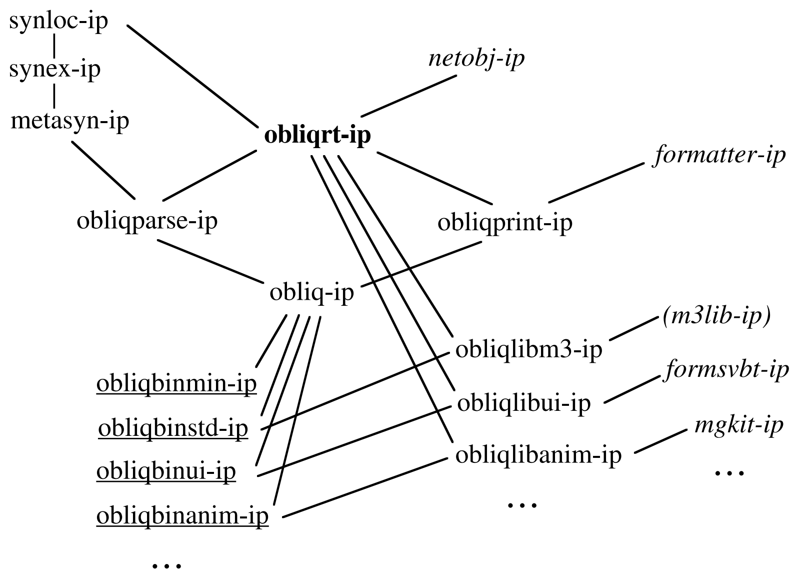
(B) if the module Foo is already loaded, Foo and all the modules that were loaded after it are erased from the top level before reloading Foo.

2.4 Building and embedding Obliq

2.4.1 The package hierarchy

One of our goals is that Obliq should be easily embedded in Modula-3 application. Although Obliq libraries adds only a very small overhead to the size of a typical Modula-3 application, we still would like to minimize the overhead.

Therefore, the Obliq implementation is split into several packages, so that only the appropriate libraries need to be linked into any given application. Another advantage of this organization, is that we can generate minimal Obliq interpreters that can act as (relatively) small network servers.



Each package has a principal interface; that interface contains a PackageSetup() routine that must be called at least once to initialize all the modules in the package.

The obliqrt-ip package implements the Obliq run-time kernel, which is the smallest part of Obliq that can be usefully embedded in an application. Note that this does not include parsers and printers; these are separately provided in obliqparse-ip and obliqprint-ip.

The obliq-ip package brings together the packages that are needed to build stand-alone Obliq interpreters. This package can be linked with various library packages to produce various flavors of Obliq interpreters.

2.4.2 The interfaces

The main client interface is obliqrt-ip/src/Obliq.i3, which refers to obliqrt-ip/src/ObTree.i3 (the parse trees) and obliqrt-ip/src/ObValue.i3 (the run-time values). Obliq.i3 contains routines to create and inspect Obliq values (including operations on remote objects), exceptions, and errors; "Eval" routines for Obliq parse trees; and sys_call registration to invoke Modula-3 routines from Obliq.

The Obliq parser and printer are separate from the run-time, and may or may not be linked into an application. The main interface to the parser is obliqparse-ip/src/ObliqParser.i3, which contains routines to parse and evaluate Obliq phrases from a reader. The interface gives an example of a simple read-eval loop. The main interface to the printer, which performs pretty-printing, is obliqprint-ip/src/ObliqPrinter.i3.

2.4.3 The libraries

Every Obliq client must link with libobliqrt-ip. The parser is in libobliqparse-ip, and the printer is in libobliqprint-ip. For building interpreters, one must link with libobliq-ip.

In every case, one must include whatever libraries are needed to get the desired Obliq built-in packages and features, as described below:

libobliqrt-ip:	array, ascii, bool, int, math, net, real, sys, text
libobliq-ip:	sys on-line extensions, on-line help
libobliqlibm3-ip:	rd, wr, lex, fmt, pickle, process, thread
libobliqlibui-ip:	color, form
libobliqlibanim-ip:	graph, zeus

2.4.4 The binaries

The "obliq" shell command is a script that runs one of four (at the moment) versions of Obliq linked with different libraries, providing different built-in packages. Network objects are supported in all versions.

The versions currently provided, along with the supported packages, are:

obliq -min	(array, ascii, bool, int, math, net, real, sys, text)	"minimal" obliq
obliq -std	(min + rd, wr, lex, fmt, pickle, process, thread)	"standard" obliq
obliq -ui	(std + color, form)	"windows" obliq
obliq -anim	(ui + graph, zeus)	"animation" obliq

By default, "obliq" means "obliq -std".

The reason for these separate versions is that the size of the binaries varies greatly depending on how many libraries are linked. The size affects link time, startup-time, and paging behavior.

A typical Obliq network server needs to be only an "obliq -min" or an "obliq -std". An Obliq network client will often be an "obliq -ui", which has a much bigger binary.

2.4.5 Embedding Obliq in an application

The appropriate client interfaces are obliqrt-ip/src/Obliq.i3, obliqparse-ip/src/ObliqParser.i3, and obliqprint-ip/src/ObliqPrinter.i3.

One may have to refer to other interfaces as well, particularly ObTree.i3 (the parser trees) and ObValue.i3 (the run-time values). Note though that ObTree.i3 is particularly specific to the current Obliq implementation, and should be used as "abstractly" as possible; the ObliqParser.i3 interface should isolate clients from any such dependencies. ObValue.i3 is also likely to evolve over time; most of its facilities can be accessed safely from Obliq.i3.

2.4.6 Extending Obliq with sys_calls

A sys_call is a cheap way of extending the functionality of an Obliq interpeter with a new "built-in" operation that invokes Modula-3 code. For more ambitious extensions, see section 2.4.7.

The interface obliqrt-ip/src/Obliq.i3 describes how to register a Modula-3 procedure so that it can be invoked from obliq. For a procedure registered under the name "foo", the Obliq syntax is:

sys_call("foo", [arg_1, ..., arg_n])

The interface obliqrt-ip/src/ObLib.i3 contains examples of how to analyze the argument array passed by Obliq to Modula-3.

Of course one must link the Modula-3 code implementing "foo" with Obliq, either in an application (section 2.4.5) or in a custom interpreter (section 2.4.8).

2.4.7 Extending Obliq with new packages

The interface obliqrt-ip/src/ObLib.i3 can be used to add a new built-in package to Obliq. One can extend Obliq new built-in types, exceptions, and operations. All the built-in Obliq packages are implemented this way.

The interface contains a detailed example of how to write and register such a package.

2.4.8 Building a customized Obliq interpreter

A new package, created as described in section 2.4.7, can be embedded into a customized Obliq interpreter. Follow the example given by obliqbinstd-ip/src/Main.m3: this is the 20-line program that builds the standard Obliq interpreter. The other obliqbin...-ip/src/Main.m3 files contain other versions of the interpreter.

3. Language reference

3.1 Data structures

3.1.1 Constants

The constants in Obliq are listed below, see also "`help syntax lexicon;`":

`ok`	a silly constant, returned by side-effecting operations
`true, false`	booleans, see "`help bool;`"
`0`, `1`, ...	integers, see "`help int;`"
`0.`, `0.1`, ...	reals, see "`help real;`"
`'a'`	chars, see "`help ascii;`"
`"abc"`	text strings, see "`help text;`"

The constant "`ok`" can be used to mean "uninitialized" in variable declarations. For future compatibility, do not use other constants for this purpose ("`0`" seems a popular choice).

3.1.2 Operators

All built-in operators are available through a set of built-in modules as qualified names (that is, as names prefixed by their originating module). For example, integer addition is "int_+(n_1,n_2)" from the "`int`" built-in module.

Frequently used built-in operations are also available without module qualification, mostly in the form of infix operators. So, "n_1+n_2" is also admitted.

Here is the list of the unqualified operators, and the built-in modules they come from.

`bool`	`is isnot not and or`
`int`	`%`
`real`	`+ - * / % > < >= <= float round`
`text`	`&`
`array`	`@`

The infix "`is`" operator is value equality on ok, booleans, integers, reals, chars, and texts, and is (network-) pointer equality on all other values; it always gives "`false`" if its two arguments are of different kind. The infix operator "`isnot`" is the negation of "`is`". The "`not`" operator is prefix, that is, it is used with ordinary function application syntax. The "`and`" and "`or`" infix operators evaluate both arguments (see also section 3.3.5 on conditional). The "`%`" operator is integer modulo. The real operators are overloaded for integers, but there are no implicit conversions between integers and reals (e.g. "`+`" takes either a pair of reals returning a real, or a pair of integers returning an integer). By special arrangement, "`0-n`" can also be written as "`-n`". The infix "`&`" operator is text concatenation. The infix "`@`" operator is array concatenation.

Every user-defined procedure of two arguments can be used in infix position.

All infix operators, both built-in and user-defined, have the same precedence, are right-associative, and evaluate their arguments from left to right. Infix operators bind less tightly than function application, object selection, and array indexing. Parentheses can be used for precedence grouping. Note that "`-5-3`" = "0`-5-3`" = "0`-(5-3)`" = "~2", while "~`5-3`" = "~8".

3.1.3 Arrays

Arrays have fixed size (once allocated), with zero-based indexing. As mentioned in the introduction, network handles to arrays can be passed over the network; therefore, arrays can be updated remotely. When iterating over a remote array, every element is accessed or updated individually over the network. Instead, subarray extraction and update involve a single communication. Subarray extraction and array concatenation produces locals copies of possibly remote arrays.

`[1,2,3,4]`	array, see "`help array;`"
`#(a)`	array size
`a[0]`	array indexing
`a[0]:=2`	array update
`a[1 for 2]`	subarray, from index 1 for length 2
`a[1 for 3]:=b`	subarray update
`a @ b`	array concatenation

Subarray update is safe even when the two arrays are the same. The subarray on the right must be at least as long as the one on the left; if it is longer, only its initial segment is used. .

3.1.4 Options

An option value is a pair of a tag (syntactically, an identifier) and a value. The tag of an option value can be tested by a case statement, discriminating among a set of expected tags. No operations other than case are defined on option values.

`option x => 3 end`	an option of tag `x` and value `3`

3.1.5 Local objects

Object behavior is described in more detail in section 1.3. Objects have the following syntax:

$\{x_1$ => a_1, ... , x_n => $a_n\}$

Protected objects (section 1.5) are enclosed in "`{|  ...  |}`" instead of curly brackets. Object operations are:

`o.x` `o.x(`a_1`, ... ,`a_n`)`	selection / invocation
`o.x:=a`	update / override
`clone(`a_1`, ... ,`a_n`)`	cloning

Cloning produces a local copy of one or more potentially remote objects.
See also section 1.6 (synchronization).

3.1.6 Network objects

The built-in "`net`" module can turn any Obliq object into a network object, by registering it with a name server. For details about the name server (which must be running), see "`man netobjd-ip`".

An object can be exported to the name server by saying:

```
- net_export("obj", "tsktsk.pa.dec.com", o);
```

where "`o`" is the object, "`tsktsk.pa.dec.com`" is the net address of the machine running the desired name server ("`"` is an abbreviation for the local machine), and "`obj`" is the registration name for the object. The object is then available through the name server, but only as long as the address space that registered it is alive; the name server does not keep the object, only a network reference to it. Only objects can be exported.

In a separate address space (or machine, domain, etc.) one can then say:

```
- let o = net_import("obj", "tsktsk.pa.dec.com");
```

Now, all object operations can be applied to this remote object (see section 1.4.2).

Object migration can be achieved by cloning a remote object locally, and replacing the remote object with a handle to the local object. This procedure migrates a single object, preserving all the (network) references accessible from the object's fields and methods. Migration must be coordinated with care to ensure that it is atomic, and that the old object is no longer referenced.

The final operation available in the "`net`" module is a net inquiry:

```
- net_who(o);
"obj@tsktsk.pa.dec.com"
```

Communication failures raise the exception "`net_failure`".

Certain Obliq built-in values make sense only in the local address space, an produce errors on any attempt to transmit them. These include threads, mutexes, conditions, processes, and forms (see appendix C). It is however easy to bundle the built-in operations for these values into objects, and then export those objects. In the case of forms, it is easy to transmit a textual form description, and generate the form remotely.

Readers and writers (appendix C.11 and C.12) can be transmitted directly over the network, because they are handled specially by the Modula-3 network object machinery. A remote reader or writer operates as an efficient network stream. Building remote reader/writer objects out of the reader/writer operations is less efficient, because buffering is then done in the wrong place.

3.3 Control structures

3.3.1 Definitions

There are three kinds of definitions, which can be used either in a local scope or at the top-level.

```
var x₁ = a₁, ..., xₙ = aₙ
let x₁ = a₁, ..., xₙ = aₙ
let rec x₁ = p₁, ..., xₙ = pₙ
```

A "`var`" definition introduces a collection of updatable variables and their initial values. A "`let`" definition introduces a collection of non-updatable identifiers and their values. A "`let rec`" definition introduces a collection of mutually recursive procedures.

In the first two cases, the terms "a_i" are all scoped in the context outside the definition. In the third case, the procedures "p_i" are scoped in the outside context extended with the variables being defined. If variables are multiply defined, the rightmost one has precedence.

Any of the three forms above can be used at the top-level, followed by a semicolon, to establish a top-level binding. See section 3.3.3 (sequencing) about local scopes.

3.3.2 Assigment

Variables introduced by "`var`" denote a storage location that can be assigned to:

```
x := a
```

The result of an assignment is the value "`ok`".

The value contained in the storage location denoted by a variable is accessed simply by mentioning the variable.

```
x := x + 1
```

As discussed in section 1.4, a variable can be a network handle.

3.3.3 Sequencing

A collection of definitions and terms (possibly causing side-effects), can be sequentially evaluated by separating the individual components by semicolons:

```
a₁; ...; aₙ
```

A final semicolon may be added.

Many syntactic contexts, such as bodies of procedures, accept sequences. But other contexts, such as argument lists, require terms. A sequence is not a term; it can be turned into a term by enclosing it in parentheses.

A sequence can be used to create a local scope, by means of definitions. The result of a sequence is the value of its last component. If the last component is a definition, then "`ok`" results.

`(var x=3; x:=x+1; x)`   yields `4`

3.3.4 Procedures and methods

Procedures and methods can be manipulated without restrictions: they can be passed as arguments, returned as results, and transmitted over the network.

proc(x_1,...,x_n) b end	a procedure term, n>=0
meth(s,x_1,...,x_n) b end	a method term, n>=0

A procedure term evaluates to a procedure closure, which is a record of the procedure term with the value of its free identifiers in the scope where it is evaluated. Similarly, a method term evaluates to a method closure.

If the free identifiers of a procedure or method denote entities with state, (updatable variables, objects, arrays), and the corresponding closure is sent over the network, then the entities with state "stay behind" and are accessed over the network when the closure is activated.

A procedure closure can be activated by an application that provides the correct number of arguments; the value of the body is then returned. A method closures must first be installed into an object, and then can be invoked via object selection. It must be given the correct number of arguments minus the self parameter; the value computed by its body is then returned.

3.3.5 Conditionals

The syntax of conditional is as shown below. There can be any number of "elsif" branches, and the "else" branch may be omitted.

if a_1 then a_2 elsif a_3 then a_4 ... else a_n end

The following boolean connectives are particularly useful in the "if" test of a conditional:

a_1 andif a_2	(* if a_1 then a_2 else false end *)
a_1 orif a_2	(* if a_1 then true else a_2 end *)

3.3.6 Case

The syntax of case is as shown below. The "else" branch may be omitted, and any "(x_i)" can also be omitted.

case a of y_1(x_1) => a_1, ..., y_{n-1}(x_{n-1}) => a_{n-1} else a_n end

The term "a" must evaluate to an option value of, say, tag t and value v. If t matches one of the "y_i", then "a_1" is executed in a scope where "x_1" (if present) is bound to v; the resulting value is the result of the case statement. If t does not match any "y_i", and the else branch is present, then "a_n" is executed and its value returned. If t does not match any "y_i", and the else branch is not present, then an error is reported.

3.3.7 Iteration

The "loop" statement repeatedly executes its body. The "exit" statement terminates the execution of the innermost loop, and causes it to return the value "ok" .

```
loop a end
exit
```

The "for" statement introduces a local identifier in the scope of its body, and iterates with the identifier ranging from the integer lower bound to the integer upper bound in increments of 1. The value "ok" is returned.

for x = a_1 to a_2 do a_3 end

The "`foreach`" statement introduces a local identifier in the scope of its body, and iterates with the identifier ranging over the elements of an array. In the "`do`" version, the values of the individual iterations are discarded, and "`ok`" returned. In the "`map`" version, those values are collected in an array that is then returned.

```
foreach x in a1 do a2 end
foreach x in a1 map a2 end
```

The "`exit`" statement can be used to terminate the innermost "`for`" or "`foreach`" statement. In the case of "`map`", a shortened array is returned containing the values of the iterations computed so far.

3.3.8 Concurrency

The primitives described in this section are built on top of, and have the same semantics as, the Modula-3 threads primitives having the same (or similar) name. The full thread interface is described in appendix C.10.

The "`mutex`" primitive returns a new mutex. The "`lock`" statements locks a mutex in a scope, returning the value of its second expression. The "`fork`" primitive starts the concurrent execution of a procedure of no arguments in a new thread, returning the thread. The "`join" primitive` waits for the termination of a thread and returns the value of the procedure it executed.

```
mutex()
lock a1 do a2 end
fork(a)
join(a)
```

The "`condition`" primitive returns a new condition. The "`signal`" and "`broadcast`" primitives wake up one or all threads, respectively, waiting on a condition. The "`wait`" primitive unlocks a mutex (first argument) until a condition is signaled (second argument), the locks the mutex again.

```
condition()
signal(a)
broadcast(a)
wait(a1,a2)
```

The "`watch`" statement is specific to Obliq's synchronized objects. It operates on the hidden mutex of such objects, thus it must occur within a method of a synchronized object.

```
watch a1 until a2 end
```

Here, "a_1" is a condition and "a_2" is a boolean expression. This statement waits for "a_2" to become true, and then terminates. Whenever "a_2" is found to be false, the statement waits for "a_1" to be signaled before trying again. The statement is equivalent to "`let` *x*`=a`$_1$`; loop if a`$_2$` then exit else wait(`*mu*`,`*x*`) end end`", where "*x*" does not occur in "a_2", and "*mu*" is the hidden mutex of the self "*s*" of the lexically enclosing method. The "`watch`" operation must be self-inflicted (section 1.3.4) with respect to such "*s*", if any, or an error is reported.

3.3.9 Exceptions

An exception is a special value that, when raised, causes unwinding of the execution stack. If the unwinding reaches the top-level, an error message is printed.

An exception is created from a string argument, which is called the exception name. Two exceptions are equal if the have equal names, hence an exception can be easily trapped in an address space different from the one in which it originated.

```
exception(a)
raise(a)
```

The unwinding of the execution stack caused by an exception can be stopped by a try-except statement, and can be temporarily suspended by a try-finally statement. The guards of a try-except statement must be exception values.

try a except a_1 => a_2, ..., a_{n-2} => a_{n-1} else a_n end
try a_1 finally a_2 end

The semantics of try statements with respect to exceptions is the same as in Modula-3. See section 3.3.10 for their behavior with respect to errors.

3.3.10 Errors

Errors, as distinct from exceptions, are produced by built-in operations in situations where a logical flaw is judged to exist in the program. These situations include divide-by-zero, array overrunning, bad operator arguments, and all cases that would produce a typechecking error in a typed language. There are no user-defined errors.

The occurrence of an error denotes a problem that should be fixed by recoding. However, errors are not show-stoppers in Obliq. Errors are intercepted (1) by the recovery clause of try-finally, after whose execution the error is reissued, and (2) by the else clause of a try-except, which can even discard the error. This way, for example, a server can log the occurrence of an infrequent internal error and restart, or can detect (to some extent) errors occurring in client-supplied procedures. Error trapping should not be used as a programming technique.

Just like exceptions, errors are returned to their originating address space. Unless something is done, an error in a server caused by a client request will propagate back to the client.

3.4 Methodology

3.4.1 Type comments

Although Obliq is an untyped language, every Obliq program, like any program, implicitly respects the type discipline in the programmer's mind. It is essential to make this discipline explicit in some way, otherwise programs quickly become unreadable and, therefore, unusable.

To this end, Obliq support a stylized form of comments that are intended to communicate type information (without enforcement). These comments are parsed according to a fixed grammar, and may appear where types usually appear in a typed language: as type definitions and as type specifications for identifiers and procedures. One need only write as much type information as is useful and convenient; type comments have no effect after parsing.

Here are examples of the syntax of "types" and their intended meaning:

`Ok`, `Bool`, `Char`, `Text`, `Int`, `Real`, `Rd`, `Wr`, `Thread`, `Mutex`, `Condition`, `Process`, `Color, Form`

conventional names for the built-in types

`X`

any identifier (capitalized by convention), for user-defined types

`X(`A_1`, …, `A_n`)`

a parameterized type, e.g. List(Int)

`?`

a run-time determined type (to be used when all else fails)

`[A]`

the type of arrays of A's

`[n*A]`

the type of arrays of A's of length n (an integer)

`(`A_1`, …, `A_n`)->A`

the type of procedures of argument types A_i ($n \geq 0$) and result type A

`(`A_1`, …, `A_n`)=>A`

the type of methods of argument types A_i ($n \geq 0$) and result type A; the type of the self argument is not included in A_i

`{`x_1`:`A_1`, …, `x_n`:`A_n`}`

the type of objects of components named x_i of type A_i

`Option `x_1`:`A_1`, …, `x_n`:`A_n` end`

the type of options with choices named x_i of type A_i

`Self(X) A`

where A is an object type, and X names the type of self for its methods. The type variable X is bound and may occur in A. This construction is mostly used for methods whose result type is the same type as the type of self.

`All(X<:A) B`

the type of polymorphic values that for any type X subtype of A produce a B; "<:A" is optional. X is bound and may occur in A

Here are examples of the usage of type comments:

`type X = A;`

a top-level type declaration. X is bound in the following scope, and may occur in A for a recursive type definition

`type X(`X_1`, …, `X_n`) = A;`

a top-level parametric type declaration. X_i are bound and may occur in A. X may occur in A, but only as $X(X_1, .., X_n)$, and in the following scope, but only as $X(A_1, .., A_n)$

`let x: A = a;`

(as opposed to let x = a) a type comment for a variable x bound by "let" (similarly for "var").

`proc(`x_1`:`A_1`, …, `x_n`:`A_n`):A, b end`

(as opposed to proc(x_1, .., x_n) b end) a commented procedure heading; any of the ":A_i" as well as ":A," may be omitted. Similarly for methods.

The value "`ok`" should be considered as having every type, so it can be used to initialize variables. However, its normal type is "`Ok`".

3.5 Built-in modules

Built-in modules provide basic operations, and the equivalent of standard Modula-3 interfaces. See Appendix C.

5. Examples

5.1 A compute server

A compute server receives a procedure with no arguments and no results, and executes the procedure at the server location via the rexec method.

```
var spy = ok; (* to spy on clients *)

let computeServer =
  net_export("ObliqComputeServer", "",
    {rexec => meth(s, p) spy:=p; p() end,
     lexec => proc(p) spy:=p; p() end
    }
```

A client may import the server and send it a procedure to execute. The procedure may have global variables at the client location:

```
let computeServer =
  net_import("ObliqComputeServer", "");

var x = 0;
computeServer.rexec(proc() x:=x+1 end);
```

The server executes the procedure, after which "`x`"="`1`" at the client, and "`spy`"="`proc() x:=x+1 end`" at the server, where the global "`x`" is still lexically bound to the client "`x`" (over the network). The server may now invoke "`spy()`", after which "`x`"="`2`" at the client.

For contrast, consider now:

```
(computeServer.lexec)(proc() x:=x+1 end);
```

This results in the server returning the procedure "`proc(p) spy:=p; p() end`" to the client. Hence, this time, the client procedure is executed at the client location. But first, the client procedure is sent over the network to the server and assigned to the "`spy`" variable there.

A compute server is an inherent security risk for the user who runs it. Compute servers should be run by Obliq interpreters lacking process and file libraries, or by interpreters running under innocuous user id's, such as "guest".

5.2 The object-oriented numerals

This rather theoretical example illustrates the expressive power of the object primitives by encoding the natural numbers as objects. Note the cloning of self.

```
let zero =
  {case =>
     proc(z,s) z() end,
   succ =>
     meth(x)
       (let o = clone(x); o.case := proc(z,s) s(x) end; o)
     end
  };
```

That's it, now some utilities:

```
let succ = proc(n) n.succ end;

let iszero =
  proc(n)
    (n.case) (proc() true end, proc(p) false end)
  end;

let pred =
  proc(n)
    (n.case) (proc() zero end, proc(p) p end)
  end;

let rec eq =
  proc(n,m)
    if iszero(n) andif iszero(m) then true
    elsif iszero(n) orif iszero(m) then false
    else pred(n) eq pred(m)
    end
  end;

let one = succ(zero);
let two = succ(one);
```

5.3 A calculator

This example illustrates method overriding, used here to store the "pending operations" of a pocket calculator.

```
let calc =
  { arg => 0.0,    (* the "visible" argument display *)
    acc => 0.0,    (* the "hidden" accumulator *)

    enter =>       (* entering a new argument *)
      meth(s, n)
        s.arg := n;
        s
      end,

    add =>         (* the addition button *)
      meth(s)
        s.acc := s.equals;
        s.equals := meth(s) s.acc+s.arg end;
        s
      end,

    sub =>         (* the subtraction button *)
      meth(s)
        s.acc := s.equals;
        s.equals := meth(s) s.acc-s.arg end;
        s
      end,

    equals =>      (* the result button (and operator stack) *)
      meth(s) s.arg end,

    reset =>       (* the reset button *)
      meth(s)
        s.arg:=0.0;
        s.acc:=0.0;
        s.equals:=meth(s) s.arg end;
        s
      end
   };
```

For example:

```
calc .reset .enter(3.5) .equals;                     (* 3.5 *)
calc .reset .enter(3.5) .sub .enter(2.0) .equals;  (* 1.5 *)
calc .reset .enter(3.5) .equals;                     (* 3.5 *)
calc .reset .enter(3.5) .add .equals;                (* 7.0 *)
calc .reset .enter(3.5) .add .add .equals;           (*10.5 *)
```

5.4 Recursion and iteration

Just to illustrate the use of recursive definition, local variables, and iteration.

```
let rec recFact =
  proc(n)
    if n is 0 then 1 else n * recFact(n-1) end;
  end;

let itFact =
  proc(n)
    var cnt = n;
    var acc = 1;
    loop
      if cnt is 0 then exit end;
      acc := cnt * acc;
      cnt := cnt - 1;
    end;
    acc;
  end;
```

5.5 The prime numbers sieve

This is an interesting example of a method overriding itself, and of an object replicating itself by cloning. The program below prints the prime numbers when the method m of the sieve object is invoked with successive integers starting from 2. Each time a new prime p is found, the sieve object clones itself into two objects. One of the clones then transforms itself into a filter for multiples of p that passes non-multiples to the other clone. At any point in time, if n primes have been printed, then there exists n filter objects plus a clone of the original sieve object.

```
let sieve =
   { m =>
        meth(s, n)
          wr_putText(wr_stdout, fmt_int(n)&"\n");
          let s0 = clone(s);
          s.m :=
            meth(s1,n1)
              if (n1 % n) is 0 then ok else s0.m(n1) end
            end;
        end
   };

(* print the primes < 100 *)
for i = 2 to 100 do sieve.m(i) end;
```

5.6 A synchronized queue

This section illustrates the use of the watch-until construct to implement a queue accessible by multiple concurrent reader and writer threads. A synchronize object contains read and write methods, which refer to global variables that are hidden from users of the queue. The object mutex is effectively used as a mutex protecting a private variable that contains the queue elements.

```
let queue =
  (let nonEmpty = condition();
   var q = [];                (* the (hidden) queue data *)

   {| write =>
        meth(s, elem)
          q := q @ [elem];    (* append elem to tail *)
          signal(nonEmpty);   (* wake up readers *)
        end,

      read =>
        meth(s)
          watch nonEmpty      (* wait for writers *)
          until #(q)>0        (* check that nobody else read *)
          end;
          let q0 = q[0];            (* get first elem *)
          q := q[1 for #(q)-1];     (* remove from head *)
          q0;                       (* return first elem *)
        end;
   |});
```

```
let t =                    (* fork a reader t, which blocks *)
  fork(proc() queue.read() end);
queue.write(3);            (* cause t to read 3 *)
join(t);                   (* get 3 from t *)
```

6. Conclusions

Appendix A: Lexicon

The ASCII characters are divided into the following classes:

Blank	*HT LF FF CR SP*
Reserved	`" ' ~`
Delimiter	`( ) , . ; [ ] _ { } ? !`
Special	`# $ % & * + - / : < = > @ \ ^ |`
Digit	`0` ... `9`
Letter	`A` ... `Z` `` ` `` `a` ... `z`
Illegal	all the others

Moreover:

a StringChar is either

- any single character that is not an Illegal character or one of (`'`), (`"`), (`\`).
- any of the pairs of characters (`\'`), (`\"`), (`\\`).

a Comment is, recursively, a sequence of non-Illegal characters and comments, enclosed between "`(*`" and "`*)`".

From these, the following *lexemes* are formed:

Space	a sequence of Blanks and Comments.
AlphaNum	a sequence of Letters and Digits starting with a Letter.
Symbol	a sequence of Specials.
Char	a single StringChar enclosed between two (`'`).
String	a sequence of StringChars enclosed between two (`"`).
Int	a sequence of Digits, possibly preceded by a single minus sign (~).
Real	two Ints separated by (.), possibly preceded by a single (~).
Delimiter	a single Delimiter character.

A stream of characters is split into lexemes by always extracting the longest prefix that is a lexeme. Note that Delimiters do not stick to each other or to other tokens even when they are not separated by Space, but some care must be taken so that Symbols are not inadvertently merged.

A *token* is either a Char, String, Int, Real, Delimiter, Identifier, or Keyword. Once a stream of characters has been split into lexemes, *tokens* are extracted as follows.

Space lexemes do not produce tokens.

Char, String, Int, and Delimiter lexemes are also tokens.

AlphaNum and Symbol lexemes are Identifier tokens, except when they have been explicitly declared to be *keywords*, in which case they are Keyword tokens. Keywords are the boldface identifiers in Appendix B.

Appendix B: Syntax overview

TOP-LEVEL PHRASES	any term or definition ended by ";"
a;	
DEFINITIONS (identifiers are denoted by "x", terms are denoted by "a")	
let x_1**=**a_1,...,x_n**=**a_n	definition of constant identifiers
let rec x_1**=**a_1,...,x_n**=**a_n	definition of recursive procedures
var x_1**=**a_1 ...,x_n**=**a_n	definition of updatable identifiers
SEQUENCES (denoted by "s")	each "a_i" (a term or a definition) is
a_1;...;a_n	executed; yields "a_n" (or "ok" if n=0)
TERMS (denoted by "a","b","c"; identifiers are denoted by "x","l")	
x	identifiers
x**:=**a	assignment
ok **true** **false** 'a' "abc" 3 1.5	constants
[a_1,...,a_n] [a for b]	arrays
a[b] a[b]**:=**c	array selection, array update
a[b for b'] a[b for b']**:=**c	subarray selection, subarray update
option l => s **end**	term "s" tagged by "l"
proc(x_1,...,x_n) s **end**	procedures
a(b_1,...,b_n)	procedure invocation
x_l(a_1,...,a_n)	invocation of "l" from module "x"
a b c	infix (right-ass.) version of "b(a,c)"
meth(x,x_1,...,x_n) s **end**	method with self "x"
{l_1**=>**a_1,...,l_n**=>**a_n}	object with fields/methods "l_1"..."l_n"
{|l_1**=>**a_1,...,l_n**=>**a_n|}	protected and synchronized object
a.l a.l(a_1, ..., a_n)	field selection / method invocation
a.l**:=**b	field update / method override
clone(a_1,...,a_n)	object cloning
d	definition
if s_1 **then** s_2	conditional
elsif s_3 **then** s_4... **else** s_n **end**	("elsif", "else" optional)
a **andif** b a **orif** b	conditional conjunction/disjunction
case s **of** l_1(x_1)**=>**s_1,...,	case over the tag "l_i" of an option value
l_n(x_n)**=>**s_n **else** s_0 **end**	binding "x_i" in "s_i" ("else" optional)
loop s **end**	loop
for i=a **to** b **do** s **end**	iteration through successive integers
foreach i **in** a **do** s **end**	iteration through an array
foreach i **in** a **map** s **end**	yielding an array of the results
exit	exit the innermost loop, for, foreach

exception("exc")	new exception value named "exc"
raise(a)	raise an exception
try s **except**	exception capture
a_1**=>**s_1,...,a_n**=>**s_n **else** s_0 **end**	("else" optional)
try s_1 **finally** s_2 **end**	finalization
condition() **signal**(a) **broadcast**(a)	creating and signaling a condition
watch s_1 **until** s_2 **end**	waiting for a signal and a boolean guard
fork(a) **join**(a)	forking and joining a thread
mutex()	creating a mutex
lock s_1 **do** s_2 **end**	locking a mutex in a scope
wait(a_1,a_2)	waiting on a mutex for a condition
(s)	block structure / precedence group

Appendix C: Built-in modules

In this appendix we list the Obliq built-in modules, which are entry points into popular Modula-3 libraries. We use an informal typing notation in the specification of the operations, including a specification of the exceptions that may be raised. Many operations raise errors as well, but these are not made explicit.

We often provide an informal English descriptions of the operations, but for ultimate details of some operations one should look at the specification of the respective Modula-3 interfaces.

The 'sys' module is special: it contains entry points into the implementation of Obliq and its computing environment.

C.1 Sys

`sys_getSearchPath(): text`
Get the current search path for 'load' and such. (Only available on-line.)

`sys_setSearchPath(t: text): ok`
Set the current search path for 'load' and such. (Only available on-line.)

`sys_print(x: t, depth: int): ok` (for all t)
Print an arbitrary value to stdout, up to some print depth. (Only available on-line.)

`sys_printText(t: text): ok`
Print a text to stdout. (Only available on-line.)

`sys_printFlush(): ok`
Flush stdout. (Only available on-line.)

`sys_pushSilence(): ok`
Push the silence stack; when non-empty nothing is printed. (Only available on-line.)

`sys_popSilence(): ok`
Pop the silence stack (no-op on empty stack). (Only available on-line.)

`sys_setPrompt(first next: text): ok`
Set the interactive prompts (defaults: first="- ", next=" "). (Only available on-line.)

`sys_getEnvVar(t: text): text`
Return the value of the env variable whose name is t, or "" if there is no such variable.

`sys_paramCount: int`
The number of program parameters.

`sys_getParam(n: int): text`
Return the n-th parameter (indexed from 0).

`sys_callFailure: exception`
Can be raised by Modula-3 code during a sys_call.

`sys_call(name: text, args: array(t)): u ! sys_callFailure` (for all t, some u)
Call a pre-registered Modula-3 procedure.

C.2 Bool

`true: bool`	
`false: bool`	
`bool_is(x: t, y: u): bool`	(for all `t,u`) (also infix `'is'`)
Equality predicate: value equality for Ok, Bool, Int, Real, Char, Text; pointer equality for the rest.	
`bool_isnot(x: t, y: u): bool`	(for all `t,u`) (also infix `'isnot'`)
Negation of `'is'`.	
`bool_not(b: bool): bool`	(also `'not(b)'`)
`bool_and(b1 b2: bool): bool`	(also infix `'and'`)
`bool_or(b1 b2: bool): bool`	(also infix `'or'`)

C.3 Int

`n: int`	(a positive integer constant)
`~n: int`	(a negative integer constant)
`int_minus(n: int): int`	
`int_+(n1 n2: int): int`	
`int_-(n1 n2: int): int`	
`int_*(n1 n2: int): int`	
`int_/(n1 n2: int): int`	
Integer divide.	
`int_%(n1 n2: int): int`	(also infix `'%'`)
Integer modulo.	
`int_<(n1 n2: int): bool`	
`int_>(n1 n2: int): bool`	
`int_<=(n1 n2: int): bool`	
`int_>=(n1 n2: int): bool`	

C.4 Real

`n.m: int`	(a positive real constant; m optional)
`~n.m: int`	(a negative real constant; m optional)
`real_minus(n: real): real`	(also `'-n'`)
`real_minus(n: int): int`	(also `'-n'`)
`real_+(n1 n2: real): real`	(also infix `'+'`)
`real_+(n1 n2: int): int`	(also infix `'+'`)
`real_-(n1 n2: real): real`	(also infix `'-'`)
`real_-(n1 n2: int): int`	(also infix `'-'`)
`real_*(n1 n2: real): real`	(also infix `'*'`)
`real_*(n1 n2: int): int`	(also infix `'*'`)
`real_/(n1 n2: real): real`	(also infix `'/'`)
`real_/(n1 n2: int): int`	(also infix `'/'`)
`real_<(n1 n2: real): bool`	(also infix `'<'`)
`real_<(n1 n2: int): bool`	(also infix `'<'`)
`real_>(n1 n2: real): bool`	(also infix `'>'`)
`real_>(n1 n2: int): bool`	(also infix `'>'`)
`real_<=(n1 n2: real): bool`	(also infix `'<='`)
`real_<=(n1 n2: int): bool`	(also infix `'<='`)
`real_>=(n1 n2: real): bool`	(also infix `'>='`)
`real_>=(n1 n2: int): bool`	(also infix `'>='`)
`real_float(n: int): real`	(also `'float(n)'`)
`real_float(n: real): real`	(also `'float(n)'`)
`real_round(n: real): int`	(also `'round(n)'`)
`real_round(n: int): int`	(also `'round(n)'`)
`real_floor(n: real): int`	
`real_floor(n: int): int`	
`real_ceiling(n: real): int`	
`real_ceiling(n: int): int`	

C.5 Math

`math_pi: real`
: 3.14159265358979323846264338 33.

`math_e: real`
: 2.71828182845904523536028747 14.

`math_degree: real`
: 0.0174532925199432957692369076 84; 1 degree in radiants.

`math_exp(n: real): real`
: *e* to the n-th power.

`math_log(n: real): real`
: log base *e*.

`math_sqrt(n: real): real`
: Square root.

`math_hypot(n m: real): real`
: sqrt((n*n)+(m*m)).

`math_pow(n m: real): real`
: n to the m-th power.

`math_cos(n: real): real`
: Cosine in radians.

`math_sin(n: real): real`
: Sine in radians.

`math_tan(n: real): real`
: Tangent in radians.

`math_acos(n: real): real`
: Arc cosine in radians.

`math_asin(n: real): real`
: Arc sine in radians.

`math_atan(n: real): real`
: Arc tangent in radians.

`math_atan2(n m: real): real`
: Arc tangent of n/m in radians.

C.6 Ascii

`c: char` (`c` a character in single quotes)

`ascii_char(n: int): char`
: The ascii character of integer code `'n'`.

`ascii_val(c: char): int`
: The integer code of the ascii character `'c'`.

C.7 Text

`t: text` (`t` a string in double quotes)

`text_new(size: int, init: char): text`
: A text of size `'size'`, all filled with `'init'`.

`text_empty(t: text): bool`
: Test for empty text.

`text_length(t: text): int`
: Length of a text.

`text_equal(t1 t2: text): bool`
: Text equality (case sensitive).

`text_char(t: text, i: int): char`
: The i-th character of a text (if it exists); zero-indexed.

`text_sub(t: text, start size: int): text`
: The subtext beginning at `'start'`, and of size `'size'` (if it exists).

`text_&(t1 t2: text): text` (also infix `'&'`)
: The concatenation of two texts.

`text_precedes(t1 t2: text): bool`
: Whether `'t1'` precedes `'t2'` in lexicographic (ascii) order.

`text_decode(t: text): text`
: Every occurrence of an escape sequence is replaced by the corresponding non-printing of formatting character: `\\` = `\`; `\'` = `'`; `\"` = `"`; `\n` = *LF*; `\r` = *CR*; `\t` = *HT*; `\f` = *FF*; `\t` = *HT*; `\`*xxx* = *xxx* (octals 000..177); `\`*c* = *c* (otherwise).

`text_encode(t: text): text`
: Every occurrence of a non-printing of formatting character is replaced by an escape sequence.

`text_explode(seps: text, t: text): array(text)`
: Split an input text into a similarly ordered array of texts, each a maximal subsequence of the input text not containing sep chars. The empty text is exploded as a singleton array of the empty text. Each sep char in the input produces a break, so the size of the result is 1 + the number of sep chars in the text. `implode(explode("c",text),'c')` is the identity.

`text_implode(sep: char, a: array(text)): text`
: Concatenate an array of texts into a single text, separating the pieces by a single sep char. A zero-length array is imploded as the empty text. `explode("c",implode('c',text))` is the identity provided that the array has positive size and sep does not occur in the array elements.

`text_hash(t: text): int`
: A hash function.

`text_findFirstChar(c: char, t: text, n: int): int`
: The index of the first occurrence of `'c'` in `'t'`, past `'n'`. `-1` if not found.

`text_findLastChar(c: char, t: text, n: int): int`
: The index of the last occurrence of `'c'` in `'t'`, before `'n'`. `-1` if not found.

`text_findFirst(p: text, t: text, n: int): int`
: The index of the first char of the first occurrence of `'p'` in `'t'`, past `'n'`. `-1` if not found.

`text_findLast(p: text, t: text, n: int): int`
: The index of the first char of the last occurrence of `'p'` in `'t'`, before `'n'`. `-1` if not found.

`text_replaceAll(old new: text, t: text): text`
: Replace all occurrences of `'old'` by `'new'` in `'t'`, as found by iterating `'findFirst'`.

C.8 Array

`[e1, ..., en]: array(t)` (for `e1...en: t`, for all `t`)

`array_new(size: int, init: t): array(t)` (for all `t`)
: An array of size `'size'`, all filled with `'init'`.

`array_#(a: array(t)): int` (for all `t`) (also `'#(a)'`)
: Size of an array.

`array_get(a: array(t), i: int): t` (for all `t`) (also `'a[i]'`)
: The i-th element (if it exists), zero-based.

`array_set(a: array(t), i: int, b: t): ok` (for all `t`) (also `'a[i]:=b'`)
: Update the i-th element (if it exists).

`array_sub(a: array(t), i n: int): array(t)` (for all `t`) (also `'a[i for n]'`)
: A new array, filled with the elements of `'a'` beginning at `'i'`, and of size `'n'` (if it exists).

`array_upd(a: array(t), i n: int, b: array(t)): ok` (for all `t`) (also `'a[i for n]:=b'`)
: Same as `'a[n+i]:=b[n]; ... ; a[i]:=b[0]'`. I.e. `'a[i for n]'` gets `'b[0 for n]'`.

`array_@(a1 a2: array(t)): array(t)` (for all `t`) (also infix `'@'`)
: A new array, filled with the concatenation of the elements of `'a1'` and `'a2'`.

C.9 Net

`net_failure: exception`

`net_who(o: t): text` (for all object types `t`)
: Return a text of the form name@server, indicating where a network object is registered, or the empty text if the object has not been registered with a name server.

`net_export(name,server: text, o: t): t ! failure` (for all object types `t`)
: Export an object under name `'name'`, to the name server at IP address `'server'`. The empty text denotes the local IP address.

`net_import(name,server: text): t ! failure` (for some object type `t`)
: Import the object of name `'name'`, from the name server at IP address `'server'`. The empty text denotes the local IP address.

C.10 Thread

```
thread_newMutex(): mutex
thread_newCondition(): condition
thread_self(): thread
thread_fork(f: ok->t, stackSize: int): thread       (for all t)
thread_join(th: thread): t                          (for some t)
thread_wait(mx: mutex, cd: condition): ok
thread_acquire(mx: mutex): ok
thread_release(mx: mutex): ok
thread_broadcast(cd: condition): ok
thread_signal(cd: condition): ok
thread_pause(r: real): ok
thread_alerted: exception
thread_alert(t: thread): ok
thread_testAlert(): bool
thread_alertWait(mx: mutex, cd: condition): ok ! alerted
thread_alertJoin(th: thread): ok ! alerted
thread_alertPause(r: real): ok ! alerted
thread_lock(m: mutex, body: ok->t): t  (for all t)
```

The lock statement.

C.11 Rd

```
rd_failure: exception
rd_eofFailure: exception
rd_new(t: text): rd
```

A Modula-3 TextRd.

```
rd_stdin: rd
```

The Modula-3 Stdio.Stdin.

```
rd_open(t: text): rd
```

A Modula-3 FileRd, open for read.

```
rd_getChar(r: rd): char
rd_eof(r: rd): bool
rd_unGetChar(r: rd): ok                             (may crash if misused!)
rd_charsReady(r: Rd): int
rd_getText(r: rd, n:int): text
rd_getLine(r: rd): text
rd_index(r: rd): int
rd_length(r: rd): int
rd_seek(r: rd, n:int): ok
rd_close(r: rd): ok
rd_intermittent(r: rd): bool
rd_seekable(r: rd): bool
rd_closed(r: rd): bool
```

C.12 Wr

```
wr_failure: exception
wr_new(): wr
```

A Modula-3 TextWr.

```
wr_toText(w: wr): text
```

For a Modula-3 TextWr.

```
wr_stdout: wr
```

The Modula-3 Stdio.Stdout.

```
wr_stderr: wr
```

The Modula-3 Stdio.Stderr.

```
wr_open(t: text): wr
```

A Modula-3 FileWr, open for write.

```
wr_openAppend(t: text): wr
```

A Modula-3 FileWr, open for append.

```
wr_putChar(w: wr, c: char): ok
wr_putText(w: wr, t: text): ok
wr_flush(w: wr): ok
wr_index(w: wr): int
wr_length(w: wr): int
wr_seek(w: wr, n:int): ok
wr_close(w: wr): ok
wr_buffered(w: wr): bool
wr_seekable(w: wr): bool
wr_closed(w: wr): bool
```

C.13 Lex

`lex_failure: exception`

`lex_scan(r: rd, t: text): text ! rd_failure`

Read from r the longest prefix formed of characters listed in t, and return it.

`lex_skip(r: rd, t: text): ok ! rd_failure`

Read from r the longest prefix formed of characters listed in t, and discard it.

`lex_match(r: rd, t: text): ok ! failure, rd_failure`

Read from r the string t and discard it; raise failure if not found.

`lex_bool(r: rd): bool ! failure, rd_failure`

Skip blanks, and attempt to read a boolean form r.

`lex_int(r: rd): int ! failure, rd_failure`

Skip blanks, and attempt to read an integer form r.

`lex_real(r: rd): real ! failure, rd_failure`

Skip blanks, and attempt to read a real form r.

C.14 Fmt

`fmt_padLft(t: text, length: int): text`

If t is shorted then length, pad t with blanks on the left so that it has the given length.

`fmt_padRht(r: rd, t: text): text`

If t is shorted then length, pad t with blanks on the right so that it has the given length.

`fmt_bool(b: bool): text`

Convert a boolean to its printable form.

`fmt_int(n: int): text`

Convert an integer to its printable form.

`fmt_real(r: real): text ! failure, rd_failure`

Convert a real to its printable form.

C.15 Process

`process_new(nameAndArgs: array(text), mergeOut: bool): proc`

Create a process from the given name and arguments. If mergeOut is true, use a single pipe for stdout and stderr.

`process_in(p: proc): wr`

The stdin pipe of a process.

`process_out(p: proc): rd`

The stdout pipe of a process.

`process_err(p: proc): rd`

The stderr pipe of a process.

`process_wait(p: proc): int`

Wait for the process to exit, close all its pipes, and return the exit code.

`process_filter(nameAndArgs: array(text), input: text): text`

Create a process from the given name and arguments, with merged output; feed the input to its stdin pipe and close it; read all the output from its stdout pipe and close it; return the output.

C.16 Form

`form_failure: exception`

`form_new(t: text): form`

Read a form description from a text.

`form_fromFile(file: text): form`

Read a form description from a file.

`form_attach(fv: form, name: text, f: form->ok): ok`

Attach a procedure to an event, under a form. The procedure is passed back the form when the event happens.

`form_getBool(fv: form, name: text): bool`

Get the boolean value of the named interactor.

`form_putBool(fv: form, name: text, b: bool): ok`

Set the boolean value of the named interactor.

`form_getInt(fv: form, name property: text): int`

Get the integer value of the named property of the named interactor. If property is the empty text, get the value property.

`form_putInt(fv: form, name property: text, n: int): ok`

Set the integer value of the named property of the named interactor. If property is the empty text, set the value property.

`form_getText(fv: form, name property: text): text`

Get the text value of the named property of the named interactor. If property is the empty text, get the value property.

`form_putText(fv: form, name property: text, t: text, append: bool): ok`

Set the text value of the named property of the named interactor. If property is the empty text, set the value property.

`form_getChoice(fv: form, radioName: text): text`

Get the choice value of the named radio interactor.

`form_putChoice(fv: form, radioName: text, choiceName: text): ok`

Set the choice value of the named radio interactor.

`form_getReactivity(fv: form, name: text): text`

Get the reactivity of the named interactor. It can be "active", "passive", "dormant", or "vanished".

`form_putReactivity(fv: form, name: text, r: text): ok`

Set the reactivity of the named interactor. It can be "active", "passive", "dormant", or "vanished".

`form_popUp(fv: form, name: text): ok`

Pop up the named interactor.

`form_popDown(fv: form, name: text): ok`

Pop down the named interactor.

`form_insert(fv: form, parent t: text, n: int): ok`

Insert the form described by t as child n of parent.

`form_delete(fv: form, parent child: text): ok`

Delete the named child of parent.

`form_deleteRange(fv: form, parent: text, n count: int): ok`

Delete count children of parent, from child n.

`form_takeFocus(fv: form, name: text, select: bool): ok`

Make the named interactor acquire the keyboard focus, and optionally select its entire text contents.

`form_show(fv: form): ok`

Show a window containing the form.

`form_hide(fv: form): ok`

Hide the window containing the form.

References