Obliq

A Language with Distributed Scope

Luca Cardelli

June 3, 1994

Abstract

Obliq is a lexically-scoped untyped interpreted language that supports distributed object-oriented computation. An Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections.

Contents

1. Introduction

Obliq is a lexically-scoped untyped interpreted language that supports distributed object-oriented computation. An Obliq computation may involve multiple threads of control within an address space, multiple address spaces on a machine, heterogeneous machines over a local network, and multiple networks over the Internet. Obliq objects have state and are local to a site. Obliq computations can roam over the network, while maintaining network connections.

1.1 Language Overview

The guiding principle that separates Obliq from other distributed procedural languages is the adherence to lexical scoping in a distributed higher-order context. This principle is conceptually simple and has a number of interesting consequences: it supports a natural and consistent semantics of distributed computation, and it enables elegant techniques for distributed programming.

In lexically scoped languages, the binding location of every identifier is determined by simple analysis of the program text surrounding the identifier. Therefore, one can be sure of the meaning of program identifiers, and can much more easily reason about the behavior of programs. In a distributed language like Obliq, lexical scoping assumes a further role. It ensures that computations have a precise meaning even when they migrate over the network: a meaning that is determined by the binding location *and* network site of identifiers, and not by execution sites.

Network-wide scoping becomes an issue in the presence of higher-order distributed computation, for example when remote sites acting as compute servers accept procedures for execution. The question here is: what happens to the free identifiers of network-transmitted procedures? Obliq takes the view that such identifiers are bound to their original locations, as prescribed by lexical scoping, even when these locations belong to different network sites.

The principal way of structuring distributed computations in Obliq is through the notion of *objects*. Network services normally accept a variety of messages; it is then natural to see each service as a *network object* (or, more neutrally, as a network interface). Obliq supports objects in this spirit, relying for its implementation on Modula-3's network objects [Birrell, *et al.* 1993b].

The Obliq object primitives are designed to be simple and powerful, with a coherent relationship between their local and distributed semantics. Obliq objects are collections of named fields, with four basic operations: selection/invocation, updating/overriding, cloning, and delegation. There are no class hierarchies, nor complex method-lookup procedures. Every object is potentially and transparently a network object. An object may become accessible over the network either by the mediation of a name server, or simply by being used as the argument or result of a remote method.

In any framework where objects are distributed across sites, it is critical to decide what to do about mobility and duplication of *state*. Normally, whenever a piece of data is transmitted from one site to another, it is implicitly copied. However, duplication of objects with state easily results in havoc, unless the state is handled consistently across sites.

To avoid problems with state duplication, objects in Obliq are local to a site and are never automatically copied over the network. In contrast, *network references* to objects can be transmitted from site to site without restrictions. An alternative approach would allow objects and their state to *migrate* from site to site, making sure that the integrity of their internal state is maintained during the act of migration. We have chosen not to support migration directly, since it requires coordination across sites, and policy decisions about the optimal time of migration. However, atomic object migration can be coded from our primitives, specifically from cloning and delegation.

In addition to the distribution of data, the distribution of computations must also be designed carefully. It is clearly desirable to be able to transmit *agents* for remote execution. However, one should not be satisfied with transmitting just the program text of such agents. Program text cannot carry with it live connections to its originating site, nor to any data or service at any other site. Hence the process of transmitting program text over the network implies a complete network disconnect from the current distributed computation. In addition, unpredictable dynamic scoping results from transmitting and then running program text containing free identifiers.

Obliq computations, in the form of procedures or methods, can be freely transmitted over the network. Actual computations (*closures*, not source text) are transmitted; lexically scoped free identifiers retain their bindings to the originating sites. Through these free identifiers, migrating computations can maintain connections to objects and locations residing at various sites. Disconnected agents can be represented as procedures with no free identifiers; these agents do not need to rely on prolonged network connectivity.

In order to concentrate on distributed computation issues and to reduce complexity, Obliq is designed as an *untyped* language. This decision leads to simpler and smaller language processors that can be easily embedded in applications. Moreover, untyped programs are somewhat easier to distribute, because we avoid problems of compatibility of types at multiple sites.

The Obliq run-time is *strongly typed*: erroneous computations produce clean errors that are correctly propagated across sites. The run-time data space is *heterogeneous*, meaning that there are different kinds of run-time values and no provisions to discriminate between them; heterogeneity discourages writing programs that would be difficult to typecheck in typed languages. Because of heterogeneity and lexical scoping, Obliq is in principle suitable for static typing. More importantly, Obliq is compatible with the disciplined approach to programming that is inspired by statically typed languages.

Lexical scoping has many interesting implications in a distributed context. One is that, together with strong run-time typing and interpreted execution, it can provide network security guarantees. Consider the situation of a server executing incoming foreign agents. Because of lexical scoping, these agents have access only to the data and resources that they can reference via free variables or that they explicitly receive in the form of procedure parameters. In particular, foreign agents cannot access data or resources at the server site that are not explicitly given to them. For example, operations on files in Obliq require file system handles that are available as global lexically bound identifiers at each site. A foreign agent can operate on the file system handle of its originating site, simply by referring to it as a free identifier. But the file system handle at the server site is outside its lexical scope, and hence unobtainable except with the cooperation of the server. Degrees of file protection can be represented by file system handles with special access rights.

1.2 Distributed Semantics

The Obliq distributed semantics is based on the notions of *sites*, *locations*, *values*, and *threads*.

Sites (that is, address spaces) contain locations, and locations contain values. Each location belongs to a unique site. We often talk about a *local site*, in relative terms, and about *remote sites*, meaning any site other than the local site. Sites are not explicit in the syntax but are implicit in operations that produce new locations.

Threads are virtual sequential instruction processors. Multiple threads may be executed concurrently, both at the same site or at different sites. A given thread may stop executing at a site, and continue executing at another site. That is, threads may jump from site to site while retaining their conceptual identity. The *current site* is where execution of a given thread of control takes place at a given moment.

In the Obliq syntax, *constant identifiers* denote values, while *variable identifiers* denote locations. A location containing a value may be updated by assignment to the variable denoting the location.

Obliq values include *basic values* (such as strings or integers), *objects*, *arrays*, *closures* (the results of evaluating methods or procedures), and other values that we need not discuss at this point.

A value may contain *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 free variables in its program text that refer to locations in the surrounding lexical scope. Basic values do not contain embedded locations. When a location is created during a computation, it is allocated at the current site.

Values may be *transmitted* over the network. A value containing no embedded locations is copied on transmission. Embedded locations are automatically replaced by network references, so that the actual locations do not move from the site where they are originally allocated. An Obliq value may contain network references to locations at different sites. In particular, a closure value may contain program text that, when executed, accesses data (bound to its free identifiers) over the network.

Every Obliq object consists of a collection of locations spanning a single site; hence the object itself is bound to a unique site[1]. This immobility of objects is not a strong limitation, because objects can be *cloned* to different sites, and because procedures can be transmitted that allocate objects at different sites. Hence, a collection of interacting objects can be dynamically allocated throughout the network, but not moved afterwards. If migration is necessary, cloning can be used to provide the needed state duplication, and delegation can be used to redirect operations to the clones.

We have stressed so far how Obliq computations can evolve into webs of network references. However, this is not necessarily the case. For example, a procedure with no free identifiers forms a completely self-contained computing *agent*. The execution of these agents may be carried out autonomously by remote compute servers (the agents may dynamically reconnect to report results). Intermediate situations are also possible, as with semi-autonomous agents that maintain low-traffic tethers to their originating site.

In conclusion, the distributed semantics of Obliq is defined so that data and computations are network-transparent: their meaning does not depend on allocation sites or execution sites (of course, computations may receive different arguments at different sites). At the same time, Obliq programs are network-aware: distribution is achieved by explicit acts that give full control on communication patterns.

Lexical scoping makes it easy to distribute computations over multiple sites, since computations behave correctly even when they are carried out at the wrong place (by some measure). Flexibility in distribution can, however, result in undesirable network traffic. Obliq relieves some of the burden of distributing data and computations, but care and planning are still required to achieve satisfactory distributed performance.

2. Local Objects

In this section we discuss the Obliq object primitives in the context of a single execution site. These primitives are then reinterpreted in the next section and given distributed meaning.

2.1 Objects and their Fields

An Obliq object is a collection of fields containing methods, aliases, or other values. A field containing a method is called a *method field*. A field containing an alias is called an *alias field*. A field containing any other kind of values, including procedure values, is called a *(proper) value field*. Each field is identified by a *field name*. Syntactically, an object has the form:

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

[1] In the implementation, network references are generated to objects and arrays, not to each of their embedded locations. However, it is consistent and significantly simpler to carry out our discussions in terms of network references to locations.

where n≥0, and ⌜x_i⌝ are distinct field names. (There is no lexical distinction between field names and program identifiers.) The terms ⌜a_i⌝ are *siblings* of each other, and the object is their *host object*. Each ⌜a_i⌝ can be any term, including a method, or an alias.

A value field is, for example:

```
x => 3
```

A method field has the form:

x => meth(y,y_1, ... ,y_n) b end

Here, the first parameter, ⌜y⌝, denotes *self*: the method's host object. The other parameters, for n>0, are supplied during method invocation. The body of the method is the term ⌜b⌝, which computes the result of an invocation of ⌜x⌝.

An alias field contains an alias:

```
x => alias y of b end
```

Operations on the ⌜x⌝ field of this object are redirected to the ⌜y⌝ field of the object ⌜b⌝. The precise effect is explained case by case in the next section.

Methods and procedures are supported as distinct concepts. Procedures start with the keyword ⌜proc⌝ instead of ⌜meth⌝ and have otherwise the same syntax. The main differences between the two are as follows. Methods can be manipulated as values but can be activated only when contained in objects, since self needs to be bound to the host object. In contrast, procedures can be activated by normal procedure call. Further, a procedure can be inserted in an object field and later recovered, while any attempt to extract a method from an object results in its activation.

Obliq methods are stored directly in objects, not indirectly in object classes or prototypes. Method lookup is a one-step process that searches a method by name within a single object. There is no class or delegation hierarchy to be searched iteratively, and there is no notion of *super*. Inheritance is obtained by cloning methods from other objects. Method lookup is implemented by a nearly constant-time caching technique, with independent caches for each operation instance, that does not penalize large objects, .

There are no provisions in Obliq for *private* fields or methods, but these can be easily simulated by lexical scoping. For example, ⌜(var x=3; { ... })⌝ is an expression setting up a local variable ⌜x⌝ and returning an object that has ⌜x⌝ in its scope. Since the scope of ⌜x⌝ is limited by the parentheses, no other part of the program can access ⌜x⌝. In addition, aliases can be used to create *views* of objects that omit certain fields or methods.

2.2 Object Operations

Apart from object creation, there are four basic operations on objects: selection/invocation, updating/overriding, cloning, and delegation. Field aliasing affects the semantics of all of them, as described below case by case.

Selection (and Invocation)

This operation has two variants for value selection and method invocation:

a.x
a.x(b_1, ... ,b_n)

The first form selects a value from the field ⌜x⌝ of ⌜a⌝ and returns it. The second form invokes a method from the field ⌜x⌝ of ⌜a⌝, supplying parameters, and returning the result produced by the method; the object ⌜`a`⌝ is bound to the self parameter of the method. For convenience, the first form can be used for invocation of methods with no parameters.

If the field ⌜`x`⌝ of ⌜`a`⌝, above, is an alias for ⌜x_1⌝ of ⌜a_1⌝, then ⌜`a.x`⌝ behaves like ⌜$a_1.x_1$⌝, and ⌜$a.x(b_1, \; ... \; ,b_n)$⌝ behaves like ⌜$a_1.x_1(b_1, \; ... \; ,b_n)$⌝. If the field ⌜$x_1$⌝ of ⌜$a_1$⌝ is itself an alias, the process continues recursively.

Updating (and Overriding)

This operation deals with both field update and method override:

```
a.x:=b
```

Here the field ⌜`x`⌝ of ⌜`a`⌝ is updated with a new value ⌜`b`⌝. If ⌜`x`⌝ contains a method and ⌜`b`⌝ is a method, we have method override. If ⌜`x`⌝ and ⌜`b`⌝ are ordinary values, we have field update. The other two possibilities are also allowed: a field can be turned into a method (of zero arguments), and vice versa.

However, if the field ⌜`x`⌝ of ⌜`a`⌝ is an alias for ⌜x_1⌝ of ⌜a_1⌝, then ⌜`a.x:=b`⌝ behaves like ⌜$a_1.x_1$:=b⌝, and so on recursively.

Cloning

The third operation is object cloning, generalized to multiple objects:

```
clone(a)
```

$$\texttt{clone}(a_1, \; ... \; ,a_n)$$

In the case of a single argument, a new object is created with the same field names as the argument object; the respective locations are initialized to the values, methods, or aliases of the argument object. Note that this operation cannot be simulated by hand, because any attempt to extract the methods or aliases of an object activates them.

In the case of multiple arguments, a single object is produced that contains the values, methods, and aliases of all the argument objects (an error is given if there are field name conflicts). Useful situations are ⌜`clone(a,{...})`⌝, where we *inherit* the fields of ⌜`a`⌝, and add new fields, and ⌜$\texttt{clone}(a_1,a_2)$⌝, where we *multiply inherit* from ⌜a_1⌝ and ⌜a_2⌝.

It is common for the parameters of ⌜`clone`⌝ to be *prototypes* (or *classes*): by convention, prototypes are objects that are meant only as repositories for methods and initial values. Via cloning, prototypes act as object generators; cloning a prototype corresponds to *newing* an object.

A *partial prototype* (or *mixin*, or *abstract class*) is a prototype whose methods refer through self to fields not in the prototype. Obviously, a partial prototype should never be used as an object or an object generator. However, one can clone partial prototypes together to obtain complete working objects.

Cloning can also be applied to objects used in computations. In particular, self can be cloned.

Delegation

Our final operation is delegation, which is the replacement of fields with aliases. In section 2.1 we have seen how to initialize alias fields:

```
{ x => alias y of b end, ... }
```

Moreover, it is possible to assign aliases to fields of existing objects with the following delegation operation (the syntax is similar to update, but this is really a separate operation):

```
a.x := alias y of b end
```

Any further operation on ⌜x⌝ of ⌜a⌝ is redirected to ⌜y⌝ of ⌜b⌝. However, delegation replaces fields with aliases regardless of whether those fields are already aliased; updating ⌜x⌝ of ⌜a⌝ again with another alias causes ⌜x⌝ of ⌜a⌝ (not ⌜y⌝ of ⌜b⌝) to be updated.

A special delegation construct can be used to delegate whole objects at once:

delegate a_1 to a_2 end

The effect is to replace every field ⌜x_i⌝ of ⌜a_1⌝ (including alias fields) with ⌜alias x_i of a_2 end⌝. Cloning can be used to assemble compound delegate objects.

Aliases and delegation must be used very carefully and, in most circumstances, are best avoided. However, delegation is already implicit in the notion of *local surrogate* of a remote object: we have simply lifted this mechanism to the language level. By doing this, we are able to put network delegation under flexible program control, as shown later in the case of object migration.

2.3 Simple Examples

Let us examine some simple examples, just to became familiar with the Obliq syntax and semantics. More advanced examples are presented in sections 4 and 5.

The following object has a single method that invokes itself through self (the ⌜s⌝ parameter). A ⌜let⌝ declaration binds the object to the identifier ⌜o⌝:

```
let o =
  { x => meth(s) s.x() end };
```

An invocation of ⌜o.x()⌝ results in a divergent computation. Divergence is obtained here without any explicit use of recursion: the self-application implicit in method invocation is sufficient.

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

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

Here are some of the operations that can be performed on ⌜o⌝:

`o.x`	Selecting the ⌜x⌝ component, producing 3.
`o.x := 0`	Setting 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 modifies its host object.

3. Remote Objects

In this section we revisit the Obliq primitives in the context of objects that are distributed over multiple sites. We discuss distributed state in general, including arrays and variables.

3.1 State

State is local in the sense that every location is forever bound to a site. At the same time, state is distributed, in the sense that there are many communicating sites. Every location at every site can potentially be accessed and modified over the network. Moreover, values may contain embedded locations belonging to current site or, via network aliases, to remote sites. Access and update of a remote location involves network communication, but is otherwise handled transparently in the same manner as access and update of a local location.

There are three kinds of entities in Obliq that directly contain locations, and hence have state:

objects: `{x₁ => a₁, ... ,xₙ => aₙ}`

every field of an object has state

access:	`a.x`,	`a.x(a₁, ... ,aₙ)`
update:	`a.x := b`,	`delegate a to b end`

arrays: `[a₁, ... , aₙ]`

every element of an array has state

access:	`a[n]`
update:	`a[n] := b`

variables: `var x = a`

variables have state (identifiers declared by ⌜`let`⌝ do not)

access:	`x`
update:	`x := b`

When objects, arrays, and variables are created during a computation, their locations are allocated at the current site.

3.2 Transmission

As discussed in the introduction, the state (i.e. set of locations) associated with objects, arrays, and variables is never duplicated or transmitted over the network. Network references to locations, however, are free to travel. Every attempted transmission of a location over the network is, in effect, intercepted and replaced by the transmission of a network reference to that location. Remote operations on these network references are reflected back to the original locations, as described in section 3.3.

Stateless values, unlike locations, are copied when transmitted over the network. Structures that are copied include basic data types and the internal representations of program text.

In the general case of transmission we may have a mixed situation, with a few layers of stateless data structures that end up referring to location. These data structures with embedded locations are copied up to the point where they refer to locations; then network references are generated.

A critical issue is the transmission of *closures*, which are the values resulting from the evaluation of procedures and methods. A closure consists of two parts: (1) the internal representation of the source text of a method or procedure, and (2) a table associating free identifiers in the source text to their values in the lexical scope of evaluation.

The free-identifiers table within a closure may refer to variables and to values with embedded locations. The general rule for transmitting structures with embedded locations applies to closures; hence closures are copied up to the locations embedded in their free-identifier tables.

For example, consider the following Obliq code, declaring a variable ⌜`x`⌝ initialized to ⌜`0`⌝, and a procedure ⌜`p`⌝ whose body refers to ⌜`x`⌝ (that is, has ⌜`x`⌝ as a free identifier):

```
var x = 0;
let p = proc() x := x+1 end;
```

Suppose that, after the execution of the first line, the variable ⌜x⌝ is bound to the location loc_0, relative to the current site s_0. Then, after the execution of the second line, the identifier ⌜p⌝ is bound to the closure:

"`proc() x := x+1 end`" *where* `x` $\equiv loc_0$

where "`proc...end`" represents the internal representation of program code, and the free identifier table is shown following *where*.

Upon transmission to a site s_1, the location loc_0 is replaced by a network reference $<s_0,loc_0>$ to that location; therefore site s_1 receives the data structure[2]:

"`proc() x := x+1 end`" *where* `x` $\equiv <s_0,loc_0>$

In general terms, a closure is a pair consisting of a piece of source text and a pointer to an evaluation stack. Transmission of a closure, in this view, implies transmission of an entire evaluation stack. The implementation of closures described above (which is well-known for higher-order languages) has the effect of reducing network traffic, by transmitting only the values from the evaluation stack that may be needed by the closure. This optimization is enabled by lexical scoping

3.3 Distributed Computation

We now reinterpret the semantics of operations on objects in the case of remote objects. In passing, we comment on the semantics of remote arrays and variables.

Selection (and Invocation)

When a value field of a remote object is selected, its value is transmitted over the network (as discussed in section 3.2) to the site of the selection.

The extraction of a remote array element and the access of a remote variable work similarly.

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

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 site, by the semantics of field selection, and then executed locally.

Updating (and Overriding)

When a field of a remote object is updated, or when a method is overridden, a value is transmitted over the network and installed into the remote object. Field update may involve the transmission of a procedure closure, and method override involves the transmission of a method closure.

The update of a remote array element and the assignment of a remote variable work similarly.

Cloning

When a collection of remote or local objects is cloned, the clone is created at the local site. Its contents (including method closures) may have to be fetched over the network.

[2] In the implementation, loc_0 is a Modula-3 network object with access and update methods.

The extraction of remote subarrays and the concatenation of remote arrays work similarly.

Delegation

In the case where the object being delegated is remote, the remote fields are replaced by the appropriate aliases. In the case where the other object is remote, aliases are generated to it.

Aliases

A local object field aliased to a remote object behaves as the field of the remote object, as described in this section case by case.

3.4 Self-inflicted Operations

The four basic object operations can be performed either as external operations on an object, or as internal operations through self. This distinction is useful in the contexts of object protection and serialization, discussed in the next two sections.

When a method operates on an object other than the method's host object, we say that the operation is *external* to the object. By contrast, when a method operates directly on its own self we say that the operation is *self-inflicted*:

> If ⌜*op*⌝ is either a select, update, clone, or delegate operation,
>
> then ⌜*op*(`o`)⌝ is *self-inflicted*
>
> iff ⌜`o`⌝ is the same object as the self of the *current method* (if any).
>
> Moreover, ⌜*op*(`o`)⌝ is *external* iff it is not self-inflicted.

Here, by the *current method* we mean the last method that was invoked in the current thread of control and that has not yet returned. Procedure calls do not change or mask the current method, even when they have not yet returned.

Whether an operation is self-inflicted can be determined by a simple run-time test. Consider, for example the object:

```
{ p => meth(s) s.q.x end,  q => ... }
```

Here the operation ⌜`s.q`⌝ is self-inflicted, since ⌜`s`⌝ is self. But the ⌜`.x`⌝ operation in ⌜`s.q.x`⌝ is self-inflicted depending on whether ⌜`s.q`⌝ returns self; in general this can be determined only at run-time.

If we replace ⌜`s.q`⌝ with a procedure call ⌜`p(s)`⌝ which simply performs ⌜`s.q`⌝, then ⌜`s.q`⌝ is still self-inflicted, and ⌜`p(s).x`⌝ may still be. The notion of "self" for self-inflicted operations is preserved through procedure calls, but not through external method invocations or thread creation.

3.5 Protected Objects

It is useful to protect objects against certain external operations, to safeguard their internal invariants. Protection is particularly important, for example, to prevent clients from overriding methods of network services, or from cloning servers. Still, protected objects should be allowed to modify their own state and to clone themselves.

This is where the notion of self-inflicted operations first becomes useful. A *protected* object is an object that rejects external update, cloning, and delegation operations, but that admits such operations when they are self-inflicted. Objects can be declared protected, as shown below:

$$\texttt{\{ protected, } \texttt{x}_1 \texttt{ => } \texttt{a}_1 \texttt{, ... , } \texttt{x}_n \texttt{ => } \texttt{a}_n \texttt{ \}}$$

Therefore, for example, methods of a protected object can update sibling fields through self, but external operations cannot modify such fields.

Note that a protection mechanism based on individual “private” fields would not address protection against cloning and delegation.

3.6 Serialized Objects

An Obliq server object can be accessed concurrently by multiple remote client threads. Moreover, local concurrent threads may be created explicitly. To prevent race conditions, it must be possible to serialize access to objects and other entities with state.

We say that an object is *serialized* when (1) in presence of multiple threads, at most one method of the object can be executing at any given time, but still (2) a method may call a sibling through self without deadlock. Note that requirement (2) does not contradict invariant (1), because an invocation through self suspends a method before activating a sibling.

The obvious approach to implementing serialized objects, adopted by many concurrent languages is to associate *mutexes* with objects (for example, see [Bal, Kaashoek, Tanenbaum 1992]). Such mutexes are locked when a method of an object is invoked, and unlocked when the method returns, guaranteeing condition (1). This way, however, we have a deadlock whenever a method calls a sibling, violating condition (2). We find this behavior unacceptable because it causes innocent programs to deadlock without good reason. In particular, an object that works well sequentially may suddenly deadlock when a mutex is added. ([Brewer, Waldspurger 1992] gives an overview of previous solutions to this problem.)

A way to satisfy condition (2) is to use reentrant mutexes, that is, mutexes that do not deadlock when re-locked by the “same” thread (for example, see [Forté 1994]). On one hand, this solution is too liberal, because it allows a method to call an arbitrary method of a different object, 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: it may make objects vulnerable to unexpected activations of their own methods, when other methods have not yet finished reestablishing the object’s invariants. On the other hand, this solution may also be too restrictive because the notion of “same” thread is normally restricted to an address space. If we want to consider control threads as extending across sites, then an implementation of reentrant locks might not behave appropriately.

We solve this dilemma by adopting an intermediate locking strategy, which we call *self serialization*, based on the notion of self-inflicted operations described in section 3.4.

Serialized objects have an implicit associated mutex, called the object mutex. An object mutex serializes the execution of selection, update, cloning, and delegation operations on its host object. Here are the simple rules of acquisition of these object mutexes:

- External operations always acquire the mutex of an object, and release it on completion.
- 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 object that is executed by the same thread. The external operation has therefore already acquired the mutex.

The serialization attribute of an object is specified as follows:

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

With self-serialization, a method can modify the state of its host object and can invoke siblings without deadlocking. A deadlock still occurs if, for example, a method invokes a method of a different object that then attempts an operation on the original serialized object. A deadlock occurs also if a method forks an invocation of a sibling and waits on the result.

Our form of object serialization solves common mutual exclusion problems, for example for network servers maintaining some simple internal state. More complex situations require both sophisticated uses of explicit mutexes, and conditional synchronization (where threads wait on *conditions* in addition to mutexes). Because of these more complex situations, Obliq supports the full spectrum of Modula-3 threads primitives [Birrell 1991; Horning, *et al.* 1993]; some through an external interface, and some directly in the syntax.

Conditional synchronization can be used also with the implicit object mutexes. A new condition ⌜*c*⌝ can be created by ⌜`condition()`⌝ and signaled by ⌜`signal(`*c*`)`⌝. A special ⌜`watch`⌝ statement allows waiting on a condition in conjunction with the implicit mutex of an object. This statement must be used inside the methods of a serialized object; hence, it is always evaluated with the object mutex locked:

```
    watch c until guard end
```

The ⌜`watch`⌝ statement evaluates the condition, and, if ⌜*guard*⌝ evaluates to true, terminates leaving the mutex locked. If the guard is false, the object mutex is unlocked (so that other methods of the object can execute) and the thread waits for the condition to be signaled. When the condition is signaled, the object mutex is locked and the boolean guard is evaluated again, repeating the process. See section 5.1 for an example.

Objects with implicit mutexes can be cloned: a fresh implicit mutex is created for the clone. Remote objects with implicit mutexes can also be cloned: a fresh implicit mutex is generated at the cloning site. Note, however, that an error is reported on any attempt to transmit an explicit mutex (or thread, or condition) between different sites, since these values are strongly site-dependent.

Consider the case of threads blocked on a condition within an object that is cloned. For local cloning, a fresh implicit mutex is created for the clone, with no threads blocked on it. The condition, however, is shared between the two objects. For remote cloning, since the watch statement refers to a condition and conditions cannot be transmitted, then the method closure that contains the watch statement cannot be transmitted, and hence the remote cloning fails.

Consider now the case of threads blocked on a condition within a method that is overridden or delegated. When the thread resumes, the original method runs to completion with a modified self. Thus, a blocked thread must deal with the fact that the self may change in non-trivial ways: this is specially insidious if the object is serialized but not protected.

Unlike objects, there is no automatic serialization for variables or arrays. If necessary, their access can be controlled through serialized objects or explicit mutexes. Even for objects, serialization is neither compulsory nor a default, since its use is not always desirable. In some cases it may be sufficient to serialize server objects (the concurrent entry points to a site) and leave all other objects unserialized.

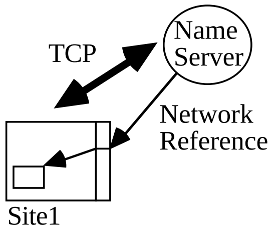
3.7 Name Servers

Obliq values can flow freely from site to site along communication channels. But such channels must first be established by interaction with a name server. A name server for Obliq programs is an external process that is uniquely identified by its IP address; it simply maintains a table associating text strings with network references [Birrell, *et al.* 1994].

The connection protocol between two Obliq sites is as follows. The first site registers a local, or remote, object under a certain name with a known name server. The second site asks the name server for (the network reference to) the object registered under that name. At this point the second site acquires a direct network reference to the object living in the first site. The name server is no longer involved in any way, except that it still holds the network reference. Obliq values and network references can now flow along the direct connection between the two sites, without having to be registered with a name server.

This protocol is coded as follows, using the built-in ⌜`net`⌝ module. An Obliq object can be exported to a name server by the command:

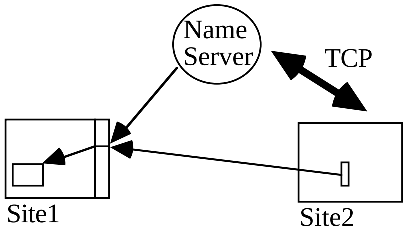
Site1: `net_export("obj",` *`NameServer`*`, site1Obj)`



where ⌜`"obj"`⌝ is the registration name for the object, ⌜`site1Obj`⌝ is the object, and ⌜*`NameServer`*⌝ is a string containing the net IP address or IP name of the machine running the desired name server. (The empty string can be used as an abbreviation for the local IP address.) The object is now available through the name server, as long as the site that exports it is alive. Objects and engines (section 3.8) are the only Obliq values that can be exported to name servers.

Any other site can then import a network reference to the object:

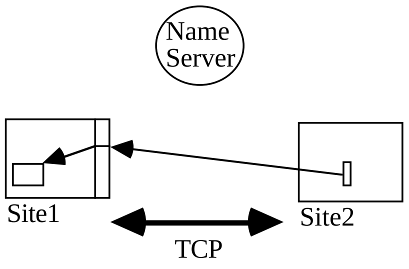
Site2: `let site1Obj = net_import("obj",` *`NameServer`*`)`



Object operations can be applied to ⌜`site1Obj`⌝ as if it were a local object, as discussed in section 3.3.

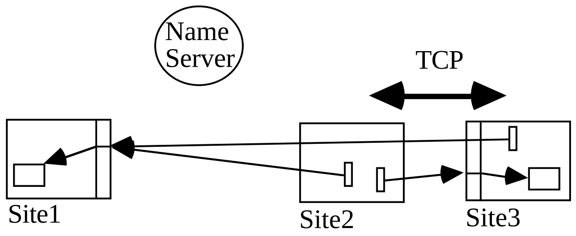
The two sites can now communicate directly; the name server is out of the loop. (It may be told to forget the object by redefining its registration name.)

Site2: `site1Obj.`*`op`*`(`*`args`*`)`



Finally, the object may be made available to a third site by transmitting it through an established communication channel:

Site2: `site3Obj.`*`op`*`(site1Obj)`



Objects are garbage collected at a site when they are no longer referred to, either locally or via network references [Birrell, *et al.* 1993a].

Another name service operation returns status information about a network reference, as a text string. It can be used to "ping" a remote object without affecting it:

```
net_who(site1Obj);
```

Communication failures raise an exception (⌜net_failure⌝), which can be trapped. These failures may mean that one of the machines involved has crashed, or that an Obliq address space was terminated. There is no automatic recovery from network failures.

3.8 Execution Engines

We shall see soon that compute servers are definable via simple network objects. However, compute servers are so common and useful that we provide them as primitives, calling them execution engines. An execution engine accepts Obliq procedures (that is, procedure closures) from the network and executes them at the engine site. An engine can be exported from a site via the primitive:

```
net_exportEngine("Engine1@Site1", NameServer, arg);
```

The ⌜arg⌝ parameter is supplied to all the client procedures received by the engine. Multiple engines can be exported from the same site under different names.

A client may import an engine and then specify a procedure to be execute remotely. An engine value behaves like a procedure of one argument:

```
let atSite1 =
  net_importEngine("Engine1@Site1", NameServer);

atSite1(proc(arg) 3+2 end);
```

Implementing engines as remote procedures, instead of a remote objects, allows self-inflicted operations to extend across sites; this turns out to be important for object migration.

4. Local Techniques

In this section we discuss a collection of single-threaded examples to illustrate Obliq's sequential features. A collection of concurrent and distributed examples is given in section 5; the impatient reader may want to skip forward. In both these sections the emphasis is on advanced, rather than tutorial, examples.

4.1 Recursion and Iteration

We start with a simple example, to illustrate the use of definitions, local variables, and control constructs. The factorial function is defined in recursive and iterative style.

```
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;
```

Identifiers are declared by ⌜let⌝, and updatable variables by ⌜var⌝. Recursive definitions are obtained by ⌜let rec⌝. The identity predicate is called ⌜is⌝. A sequence of statements separated by semicolons returns the value of the last statement; hence the iterative factorial program returns ⌜acc⌝.

4.2 The Object-Oriented Numerals

This next example illustrates the expressive power of the object primitives by encoding the natural numbers purely in terms of objects.

```
let zero =
  {case =>
     proc(pz,ps) pz() end,
   succ =>
     meth(self)
       let o = clone(self);
       o.case := proc(pz,ps) ps(self) end;
       o
     end};
```

The numeral ⌜zero⌝ has two fields. The ⌜succ⌝ field produces successive numerals by appropriately modifying the current numeral. The ⌜case⌝ field is used to discriminate on zero: the idiom ⌜(n.case)(proc() b end, proc(p) c end)⌝ is read, informally, as "if n is zero then return b, else bind the predecessor of n to p and return c".

The code of the ⌜succ⌝ method depends heavily on Obliq peculiarities: it clones self, and embeds the current self into a procedure closure, so that it can be used later. For example, the numeral ⌜one⌝, computed as, ⌜zero.succ()⌝, is:

```
 {case => proc(pz,ps) ps(zero) end,
  succ => (as for zero) }
```

Hence, ⌜one.case(pz,ps)⌝ correctly applies ⌜ps⌝ to the predecessor of ⌜one⌝.

To show that the encoding is fully general, we define the successor, predecessor, and test for zero procedures:

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

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

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

4.3 The Prime Numbers Sieve

This example shows an interesting case of methods overriding themselves, and of objects replicating themselves 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; non-multiples are passed to the other clone.

```
let sieve =
   { m =>
        meth(s, n)
          print(n);              (* defined elsewhere *)
          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;
```

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.

4.4 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 *)
```

4.5 Surrogates

Here we create a non-trivial surrogate for the calculator object of section 4.4. Unlike the original calculator, this object is protected against outside interference. Some of the calculator fields are shared by aliasing, some are hidden, some are renamed, and one is added.

```
let publicCalc =
   { protected,
     enter => alias enter of calc end,
     pi => meth(s) s.enter(3.141592653589793238462643383) end,
     plus => alias add of calc end,
     minus => alias sub of calc end,
     equals => alias equals of calc end,
     reset => alias reset of calc end }
```

5. Distributed Techniques

In this section we code some distributed programming techniques in Obliq. Each example is typical of a separate class of distributed programs, and illustrates the unique features of Obliq.

5.1 A Serialized Queue

We begin with an example of ordinary concurrent programming to illustrate the threads primitives that are used in the sequel. We implement a queue that can be accessed consistently by concurrent reader and writer threads.

The queue is implemented as a serialized object with ⌜`read`⌝ and ⌜`write`⌝ methods. These methods refer to free identifiers that are hidden from users of the queue. The object mutex is used, implicitly, to protect a private variable that contains an array of queue elements. Another private variable contains a *condition* ⌜`nonEmpty`⌝ used for signaling the state of the queue.

The write method adds an element to the queue, and *signals* the non-empty condition, so that at least one reader thread waiting on that condition wakes up (a similar *broadcast* operation wakes up all waiting threads). The object mutex is locked throughout the execution of the write method, therefore excluding other writer or reader threads.

When a read method starts executing, the object mutex is locked. Its first instruction is to watch for the non-empty condition, and for the existence of elements in the queue. If the queue is non-empty, the reader simply goes ahead and removes one element from the queue. If the queue is empty, the reader thread is suspended and the object mutex is released (allowing other reader and writer threads to execute). The reader is suspended until it receives a signal for the non-empty condition; then the object mutex is locked, and the reader thread proceeds as above (possibly being suspended again if some other reader thread has already emptied the queue).

What is important here is that a reader thread may be blocked inside a method, and yet a writer thread can get access and eventually allow the first thread to proceed. Hence, even though only one thread at a time can run, multiple threads may be simultaneously present "in" the object.

Here, ⌜`[...]`⌝ is an array, ⌜`#`⌝ is array-size, and ⌜`@`⌝ is array-concatenation.

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

   {protected, serialized,
      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 size of queue *)
          end;
          let q0 = q[0];           (* get first elem *)
          q := q[1 for #(q)-1];    (* remove from queue *)
          q0;                      (* return first elem *)
        end;
   });
```

Let us see how this queue can be used. Suppose a reader is activated first when the queue is still empty. To avoid an immediate deadlock, we fork a thread running a procedure that reads from the queue; this thread blocks on the ⌜watch⌝ statement. The reader thread is returned by the ⌜fork⌝ primitive, and bound to the identifier ⌜t⌝:

```
let t =                  (* fork a reader t, which blocks *)
  fork(proc() queue.read() end, 0);
```

Next we add an element to the queue, using the current thread as the writer thread. A non-empty condition is immediately signaled and, shortly thereafter, the reader thread returns the queue element.

```
queue.write(3);          (* cause t to read 3 *)
```

The reader thread has now finished running, but is not completely dead because it has not delivered its result. To obtain the result, the current thread is joined with the reader thread:

```
let result = join(t);    (* get 3 from t *)
```

In general, ⌜join⌝ waits until the completion of a thread and returns its result.

5.2 Compute Servers

The compute server defined below receives a client procedure ⌜p⌝ with zero arguments via the ⌜rexec⌝ method, and executes the procedure at the server site. This particular server cheats on clients by storing the latest client procedure into a global variable ⌜replay⌝. Another field, ⌜lexec⌝, is defined similarly to ⌜`rexec`⌝, but ⌜`rexec`⌝, is a method field, while ⌜`lexec`⌝, is a value field containing a procedure value: the operational difference is discussed below.

```
(* Server Site *)
var replay = proc() end;

net_export("ComputeServer", NameServer,
  {rexec => meth(s, p) replay:=p; p() end,
   lexec => proc(p) replay:=p; p() end})
```

A client may import the compute server and send it a procedure to execute. The procedure may have free variables at the client site; in this example it increments a global variable ⌜`x`⌝:

```
(* Client Site *)
let computeServer =
  net_import("ComputeServer", NameServer);

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

When the server executes its ⌜`rexec`⌝ method, ⌜`replay`⌝ is set to (a closure for) ⌜`proc() x:=x+1 end`⌝ at the server site, and then ⌜`x`⌝ is set to ⌜`1`⌝ at the client site, since the free ⌜`x`⌝ is lexically bound to the client site. Any variable called ⌜`x`⌝ at the server site, if it exists, is a different variable and is not affected. At the server we may now invoke ⌜`replay()`⌝, setting ⌜`x`⌝ to ⌜`2`⌝ at the client site.

For contrast, consider the execution of the following line at the client site:

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

This results in the server returning the procedure ⌜`proc(p) replay:=p; p() end`⌝ to the client, by the semantics of remote field selection, with ⌜`replay`⌝ bound at the server site. Then the client procedure ⌜`proc() x:=x+1 end`⌝ is given as an argument. Hence, this time, the client procedure is executed at the client site. Still, the execution at the client site causes the client procedure to be transmitted to the server and bound to the ⌜`replay`⌝ variable there. The final effect is the same.

5.3 A Database Server

This example describes a simple server that maintains a persistent database of “fortunes”. Each client may add a new fortune via a ⌜`learn`⌝ method, and may retrieve a fortune entered by some client via a ⌜`tell`⌝ method. The server handles concurrent client access, and saves the database to file to preserve data through outages. An initial empty database is assumed.

The built-in libraries for readers (⌜`rd_`⌝), writers (⌜`wr_`⌝), and data storage (⌜`pickle_`⌝) are described in section B.6.

```
let writeDB =
  proc(dB)
    let w = wr_open(fileSys, "fortune.obq");
    pickle_write(w, dB); wr_close(w)
  end;

let readDB =
  proc()
    let r = rd_open(fileSys, "fortune.obq");
    let a = pickle_read(r); rd_close(r); a
  end;

var i = -1;

let fortune =
  {protected, serialized,
     dB => readDB(),

     tell =>
       meth(self)
         if #(self.dB) is 0 then "<bad luck>"
         else
           i := i+1;
           if i >= #(self.dB) then i:=0 end;
           self.dB[i]
         end
       end,

     learn =>
       meth(self, t)
         self.dB := self.dB @ [t];
         writeDB(self.dB);
       end,
  };

net_export("FortuneServer", NameServer, fortune);
```

5.4 Remote Agents

Compute servers (section 5.2) and execution engines (section 3.8) can be used as general object servers; that is, as ways of allocating objects at remote sites. These objects can then act as *agents* of the initiating site.

Suppose, for example, that we have an engine exported by a database server site. The engine provides the database as an argument to client procedures:

```
  (* DataBase Server Site *)
  net_exportEngine("DBServer", NameServer, dataBase);
```

A database client could simply send over procedures performing queries on the database (which, for complex queries, would be more efficient than repeatedly querying the server remotely). However, for added flexibility, the client can instead create an object at the server site that acts as its remote agent:

```
(* DataBase Client Site *)
let atDBServer =
  net_importEngine("DBServer", NameServer);

let searchAgent =
  atDBServer(
    proc(dataBase)
      {state => ...,
       start => meth ... end,
       report => meth ... end,
       stop => meth ... end}
    end);
```

The execution of the client procedure causes the allocation of an object at the server site with methods ⌜`start`⌝, ⌜`report`⌝, and ⌜`stop`⌝, and with a ⌜`state`⌝ field. The server simply returns a network reference to this object, and is no longer engaged.

We show below an example of what the client can now do. The client starts a remote search in a background thread, and periodically request a progress report. If the search is successful within a given time period, everything is fine. If the search takes too long, the remote agent is aborted via ⌜`stop`⌝. If an intermediate report proves promising, the client may decide to wait for however long it takes for the agent to complete, by joining the background thread.

```
(* DataBase Client Site *)
let searchThread =
  fork(proc() searchAgent.start() end, 0);

var report = "";
for i = 1 to 10 do
  pause(6.0);
  report := searchAgent.report();
  if successful(report) then exit end;
  if promising(report) then
    report := join(searchThread); exit;
  end;
end;
searchAgent.stop();
```

Client resources at the server site are released when the client garbage collects the search agents, or when the client site dies [Birrell, *et al.* 1993a].

This technique for remotely allocating objects can be extended to multiple agents searching multiple databases simultaneously, and to agents initiating their own sub-agents.

5.5 Application Partitioning

The technique for remotely allocating objects described in section 5.4 can be used for *application partitioning*. An application can be organized as a collection of procedures that return objects. When the application starts, it can pick a site for each object and send the respective procedure to a remote engine for that site. This way, the application components can be (initially) distributed according to dynamic criteria.

5.6 Agent Migration

In this example we consider the case of an untethered agent that moves from site to site carrying along some state[White 1994]. We write the state as an object, and the agent as a procedure parameterized on the state and on a site-specific argument:

```
let state = { ... };
let agent = proc(state, arg) ... end;
```

To be completely self-contained, this agent should have no free identifiers, and should use the state parameter for all its long-term memory needs.

The agent can be sent to a new site as follows, assuming ⌜`atSite1`⌝ is an available remote engine:

```
atSite1(proc(arg) agent(copy(state),arg) end)
```

The ⌜`copy`⌝ operation is explained below, but the intent should be clear: the agent is executed at the new site, with a local copy of the state it had at the previous site. The agent's state is then accessed locally at the new site. Implicitly, we assume that the agent ceases any activity at the old site. The agent can repeat this procedure to move to yet another site.

The ⌜`copy`⌝ operation is a primitive that produces local copies of (almost) arbitrary Obliq values, including values that span several sites. Sharing and circularities are preserved, even those that span the network. Not all values can be copied, however, because not all values can be transmitted. Protected objects cause exceptions on copying, as do site-specific values such as threads, mutexes, and conditions.

This techniques allows autonomous agents to travel between sites, perhaps eventually returning to their original site with results. The original site may go off-line without directly affecting the agent.

The main unpleasantness is that, because of copying, the state consistency between the old site and the new site must be preserved by programming convention (by not using the old state). In the next section we see how to migrate state consistently, for individual objects.

5.7 Object Migration

This example uses a remote execution engine to migrate an object between two sites. First we define a procedure that, given an object, the name of an engine, and a name server, migrates the object to the engine's site. Migration is achieved in two phases: (1) by causing the engine to remotely clone the object, and (2) by delegating the original object to its clone.

```
let migrateProc =
  proc(obj, engineName)
    let engine = net_importEngine(engineName, NameServer);
    let remoteObj = engine(proc(arg) clone(obj) end);        (1)
    delegate obj to remoteObj end;                           (2)
```

```
        remoteObj;
      end;
```

After migration, operations on the original object are redirected to the remote site, and executed there.

It is critical, though, that the two phases of migration be executed atomically, to preserve the integrity of the object state[3]. This can be achieved by serializing the migrating object, and by invoking the ⌜migrateProc⌝ procedure from a method of that object, where it is applied to self:

```
let obj1 =
  { serialized, protected,
    ...        (other fields)
    migrate =>
      meth(self, engineName)
        migrateProc(self, engineName);
      end};

let remoteObj1 = obj1.migrate("Engine1@Site1")
```

Because of serialization, the object state cannot change during a call to ⌜migrate⌝. The call returns a network reference to the remote clone that can be used in place of ⌜obj1⌝ (which, anyway has been delegated to the clone).

We still need to explain how migration can work for protected objects, since such objects are protected against external cloning and delegation. Note the ⌜migrateProc(self, ...)⌝ call above, where ⌜self⌝ is bound to ⌜obj1⌝. It causes the execution of:

```
engine(proc(arg) clone(obj1) end)
```

Rather subtly, the cloning of ⌜obj1⌝ here is self-inflicted (section 3.4), even though it happens at a site different from the site of the object. According to the general definition, ⌜clone(obj1)⌝ is self-inflicted because ⌜obj1⌝ is the same as the self of the last active method of the current thread, which is ⌜migrate⌝. The delegation operation is similarly self-inflicted. Therefore, the protected status of ⌜obj1⌝ does not inhibit self-initiated migration.

Migration permanently modifies the original object, redirecting all operations to the remote clone. In particular, if ⌜obj1⌝ is asked to migrate again, the remote clone will properly migrate.

We now make the example a bit more interesting by assuming that the migrating object ⌜obj1⌝ is publicly available through a name server. The ⌜migrate⌝ method can register the migrated object with the name server under the old name:

```
let obj1 =
  net_export("obj1", NameServer,
    { serialized, protected,
      ...
      migrate =>
        meth(self, engineName)
          net_export("obj1", NameServer,
            migrate(self, engineName));
```

[3] “Captain, we have a problem. We teleported an instance of yourself successfully to the planet. But you here failed to disintegrate. This is most unfortunate; if you could just step into this waste recycler ...”

```
                    end};
```

This way, old clients of ⌜obj1⌝ go through aliasing indirections, but new clients acquiring ⌜obj1⌝ from the name server operate directly on the migrated object.

5.8 Application Servers

Visual Obliq [Bharat, Brown 1994] is an interactive distributed-application and user-interface generator, based on Obliq. All distributed applications built in Visual Obliq follow the same model, which we may call the application server model. In this model, a centralized server supplies interested clients, dynamically, with both the client code and the client user interface of a distributed application. The code transmitted to each client retains lexical bindings to the server site, allowing it to communicate with the server and with other clients. Each client may have separate local state, and may present a separate view of the application to the user. A typical example is a distributed tic-tac-toe game.

6. Syntax Overview

TOP-LEVEL PHRASES		any term or definition ended by ⌜;⌝
a;		
DEFINITIONS (denoted by ⌜d⌝; 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⌝; libraries are denoted by ⌜m⌝)		
x	m_x	identifiers
x:=a		assignment
ok **true** **false** 'a' "abc" 3 1.5		constants
[a_1,...,a_n]		arrays
a[b]	a[b]:=c	array selection, array update
a[b_1 for b_2]	a[b_1 for b_2]:=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
m_x(a_1,...,a_n)		invocation of ⌜x⌝ from library ⌜m⌝
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 named ⌜l_1⌝...⌜l_n⌝
{**protected**, **serialized**, ...}		protected and serialized object
{l_1**=>alias** l_2 **of** a_2 **end**,...}		object with delegated fields
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

a_1.l_1**:=alias** l_2 **of** a_2 **end**	field delegation
delegate a_1 **to** a_2 **end**	object delegation
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_1,a_2) **join**(a)	forking and joining a thread
pause(a)	pausing the current 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

7. Conclusions

Obliq addresses a very dynamic form of distributed programming, where objects can delegate their behavior over the network, and where computations can roam between network sites. We feel that this kind of programming is still in its infancy, and that not all the fundamental issues can yet be addressed at once. Where in doubt, we have given precedence to flexible mechanism over robust methodology, hoping that methodology will develop with experience. In this spirit, for example, Obliq could be used to experiment in the design and implementation of agent/place paradigms [White 1994], using the basic techniques of section 5.

Related Work

Obliq's features and application domains overlap with programming languages such as ML [Milner, Tofte, Harper 1989; Reppy 1991], Modula-3 [Nelson 1991], and Self [Ungar, Smith 1987], with scripting languages such as Tcl [Ousterhout 1994], AppleScript [Apple 1993], VBA [Brockschmidt 1994; Mansfield 1994], and Telescript [White 1994], and with distributed languages such as Orca [Bal, Kaashoek, Tanenbaum 1992], Forté [Forté 1994], and Facile [Thomsen, *et al.* 1993].

None of these languages, however, has the same mix of features as Obliq, particularly concerning the distribution aspects. Our choice of features was largely determined by the idea of a distributed lexically scoped language, by the desire for a simple object model that would scale up to distributed computation, and by the availability of a sophisticated network-objects implementation technology.

The Obliq object primitives were designed in parallel with work on the type theory of objects [Abadi, Cardelli 1994]; distributed scoping and distributed semantics, however, are not treated there.

Status

Obliq has been available at Digital SRC for about a year. In addition to incidental programming, it has been used extensively as a scripting language for algorithm animation [Brown 1994] and 3D graphics [Najork, Brown 1994], and as the basis of a distributed-application builder (Visual Obliq [Bharat, Brown 1994]).

The Obliq implementation provides access to many popular Modula-3 libraries [Horning, *et al.* 1993] and to an extensive user interface tool kit [Brown, Meehan 1994]. Obliq can be used as a stand-alone interactive interpeter. It can also be embedded as a library in Modula-3 applications, allowing them to interact remotely through Obliq scripts.

The implementation and complete documentation is available on the World Wide Web at ˹http://www.research.digital.com/SRC/home.html˺.

Future Work

Issues of authentication, security, authority delegation, and accounting remain to be explored.

Acknowledgments

The Network Objects project at SRC provided the infrastructure without which Obliq would never have been thought of. Alan Knaff implemented the metaparser layer used in the Obliq parser.

A. Language Reference

This section describes the syntax and semantics of the Obliq language. Interactions with the surrounding system environment are described in section B. Interactions with the surrounding programming environment are described in section C.

A.1 Syntactic Structures

We begin with an overview of some principles that pervade the syntax of Obliq. While the formal grammar has the final word (section A.6), these principles should help in predicting the correct syntax to be used in programs.

Obliq's syntactic structures can be classified into *identifiers*, *definitions*, *terms*, and *term sequences*. Definitions establish bindings, terms denote values, and term sequences represent sequential evaluation. *Final commas* in term and definition lists, as well as *final semicolons* in term sequences, are always optional.

A.1.1 Identifiers

Obliq's *unqualified identifiers* are either case sensitive sequences of alphanumerics beginning with a letter, or sequences of special characters (section A.5). By convention, identifiers used for constants, variables, procedures, fields, and methods begin with a lower case letter, and are internally capitalized on word boundaries. Type identifiers (section A.4.1) begin with an upper case letter.

Qualified identifiers have the form ⌜`m_x`⌝ where ⌜`m`⌝ is a *library* name (alphanumeric), and ⌜`x`⌝ is an unqualified identifier. By convention, the names of built-in libraries begin with lower case letters, while the names of user libraries begin with an upper case letter.

All identifiers are *lexically scoped*. Unqualified identifiers are subject to block scoping, while library names are scoped in a global environment.

Field names (for object and option values) have the same lexical structure as unqualified identifiers. Field names are not subject to scoping.

A.1.2 Definitions

Definitions begin with either ⌜`let`⌝, ⌜`let rec`⌝, or ⌜`var`⌝, followed by a comma-separated list of binders, which bind unqualified identifiers to terms. A ⌜`let`⌝ definition introduces constant identifiers, while a ⌜`var`⌝ definition introduces assignable identifiers (variables). A ⌜`let rec`⌝ definition introduces a collection of identifiers bound to mutually recursive procedures.

A.1.3 Terms

The Obliq language is value-oriented: almost every syntactic structure is a *term*, and every term produces a value. Terms whose main purpose is to cause side-effects produce the value ⌜`ok`⌝. Terms can be classified into *identifier terms*, *data terms*, *constructs*, and *operations*.

Identifier terms are qualified or unqualified identifiers.

Data terms have specialized syntax for various built-in data structures.

Constructs have individual specialized syntax, but whenever they begin with a keyword they end with the keyword ⌜`end`⌝.

Operations can be either prefix or infix. A prefix operation consists of an term (indicating an operation, or evaluating to a procedure) followed by a parenthesized, comma-separated, list of argument terms. An infix operation consists of a term, an unqualified identifier, and another term. Every unqualified identifier that denotes a built-in binary operator or a binary procedure can be used with both prefix and infix syntax. The operator ⌜-⌝ (minus) can be simply placed in front of a term, without requiring parentheses.

A.1.4 Term Sequences

Term sequences are lists of terms separated by semicolons: they indicate the sequential execution of terms from left to right. Semicolons are used in Obliq exclusively to indicate sequential execution; all other kinds of lists are separated by commas.

Definitions happen to be terms as well (their value is always the constant ⌜ok⌝), and hence may appear in sequences. Definitions establish bindings whose scope extends to the whole sequence to their right.

A.1.5 Built-In Operators

All built-in operators are available as qualified names through a set of built-in libraries. For example, real addition is ⌜real_+(r_1,r_2)⌝ from the ⌜real⌝ built-in library. Common built-in operations are made available also without library qualification, mostly in the form of infix operators. So, ⌜r_1+r_2⌝ is also admitted.

A.1.6 Operator Precedence

Operator precedence is the same for all infix operators, both built-in and user-defined. All operators are right-associative, and evaluate their arguments from left to right. Infix operators bind less tightly than procedure call, object selection, and array indexing. Parentheses can be used for precedence grouping.

The minus sign for negative number literals is ⌜~⌝; this is not an operator: it is part of the literal. The form ⌜-n⌝ is equivalent to ⌜0-n⌝, particularly with respect to operator precedence. As a consequence of these rules, ⌜-5-3⌝ ≡ ⌜0-5-3⌝ ≡ ⌜0-(5-3)⌝ = ⌜~2⌝, while ⌜~5-3⌝ = ⌜~8⌝.

A.2 Data Structures

A network address is a pair consisting of a *site address* and a *memory address* at that site. The semantics of Obliq data can be described consistently by considering all addresses as network addresses in the sense above. Obliq data structures are assembled out of network addresses, just like ordinary data structures are assembled out of local addresses (more precisely, the implementation is designed to create this illusion). With this proviso, Obliq data structures can be discussed with almost no reference to the existence of multiple sites.

A.2.1 Value Identity

A value is a data structure that is the result of an Obliq computation. Values may *share* substructures. Updates to shared substructures may be visible from separate value roots. To understand when and how sharing occurs, it is critical to know under what circumstances two Obliq values are *identical*. The entire network semantics of Obliq can be glimped by the details of this definition.

The infix operator ⌜is⌝ determines value identity. It returns a boolean on every pair of arguments, including pairs of different types. Its negation is the operator ⌜isnot⌝:

`a is b`	is a identical to b?
`a isnot b`	is a not identical to b?

A value maintains its identity as long as it is not copied: copying a value produces a *similar* value which is not identical to it. For the basic types (ok, booleans, integers, reals, chars, texts, and exceptions), we imagine that there is a single instance of each value, which is never copied. For other types, values are copied by specific operations, such as object cloning and array concatenation, and by network transmission.

Most importantly, values are *not* copied on identifier definition and access, on local assignment and update, or on local parameter passing and result. In these situations, a value may become a shared substructure of two or more other structures. Values with state (objects and arrays) are not copied even on remote versions of the situation above.

Let us spell out the consequences for ⌜`is`⌝. For basic types the ⌜`is`⌝ predicate corresponds to semantic value equality. For example, an integer is another integer if they are the same number, and a text is another text if they contain the same sequence of characters.

For objects and arrays, the ⌜`is`⌝ predicate corresponds to equality of the network addresses where the actual objects and arrays (not their network references) are stored.

For most other types (options, closures, readers, and writers), the ⌜`is`⌝ predicate corresponds to equality of the local addresses where the values are stored.

Finally, certain data types make sense only within a site (local threads, mutexes, conditions, processes, forms); network transmission of these values is inhibited. These values are identical when they are stored at the same local address.

A.2.2 Constants

The constants literals are listed below, see section A.5 for the lexical details.

`ok`	a trivial constant, returned by side-effecting operations
`true, false`	booleans, see section B.6.2.
`0`, `1`, `~1`, ...	integers, see section B.6.3.
`0.`, `0.1`, `~0.1`, ...	reals, see sections B.6.4 and B.6.5.
'`a`'	chars, see section B.6.6.
`"abc"`	text strings, see section B.6.7.

The constant ⌜`ok`⌝ can be used to mean “uninitialized” in variable declarations. For characters and strings, escape sequences (`\\`, `\'`, `\"`, `\n`, `\r`, `\t`, `\f`, `\`*`xxx`* for *`xxx`* octal) are supported with the usual meaning (section A.5).

A.2.3 Operators

Here is the list of all the predefined unqualified operators. On the left, we list the built-in libraries they belong to. For the list of all built-in libraries (and hence of all qualified and unqualified operators), see section B.6. Operators evaluate all their arguments from left to right.

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

The ⌜`not`⌝ operator is prefix (that is, its argument must be parenthesized). The ⌜`and`⌝ and ⌜`or`⌝ infix operators evaluate both arguments (but see also section A.3.5). These operators accept only boolean arguments.

The infix ⌜`%`⌝ operator is integer modulo.

The operators on real numbers are overloaded with corresponding operators on integers. The infix arithmetic operators on reals accept also pairs of integers and return an integer, but do not accept mixed integer-real arguments. The infix comparison operators on reals similarly accept a pair of integer arguments, but not mixed arguments. The prefix operators ⌜`float`⌝ and ⌜`round`⌝ accept both integers and reals. The form ⌜`-n`⌝ is equivalent to ⌜`0-n`⌝.

The infix ⌜`&`⌝ operator is text concatenation.

The prefix ⌜`#`⌝ operator is array size; the infix ⌜`@`⌝ operator is array concatenation.

A.2.4 Arrays

Arrays have fixed size (once allocated), with zero-based indexing.

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

All array operations are bound-checked. When the array is remote, each indexing and update operation causes a network communication.

Subarray extraction and array concatenation produce local copies of possibly remote arrays. Note that array values are always shared, unless explicitly copied by these two operations (or copied element by element).

Subarray extraction, subarray update, and array concatenation cause at most one network communication for each argument.

Subarray update operates correctly even when updating overlapping segments of the same array. The source array must be at least as long as the destination array; if it is longer, only its initial segment is used.

See also section B.6.8, which includes operations to initialize arrays from values and iterators.

A.2.5 Options

An option value is a pair of a tag (syntactically, an identifier) and a value. Such a tag can be tested by a case statement, which discriminates between a set of expected tags. No operation other than case is defined on option values.

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

A.2.6 Objects

Objects are collections of *fields* ⌜x_i => a_i⌝, where ⌜x_i⌝ is a *field name*, and ⌜a_i⌝ is a term. A *method field* is a field that contains a method closure. An *alias field* is a field that contains an alias. Otherwise, a field is called a *value field*.

{x_1 => a_1, ... ,x_n => a_n}	for $n \geq 0$

Objects may have two *attributes*: *protected* and *serialized* (section A.2.7) The keywords ⌜`protected`⌝ and/or ⌜`serialized`⌝ may be placed after the left brace, each optionally followed by a comma.

An aliased field denotes a field within another object. Most operations on aliases are redirected to the fields they denote, as described in section A.2.6.

{x$_1$ => alias x of a end, ... }   an alias for field x of object a

An error is produced if the object ⌜a⌝ does not have the field ⌜x⌝.

We now describe the primitive operations on objects.

Selection

a.x

If ⌜x⌝ is a value field, then the value is returned. If ⌜x⌝ is a method field containing a method of no arguments, then the method is invoked by supplying a as its first parameter, and its result (or error, or exception) is returned. If ⌜x⌝ is an alias field for ⌜x$_0$⌝ of ⌜a$_0$⌝, then ⌜a$_0$.x$_0$⌝ is executed. Selection fails if ⌜x⌝ is not a field of ⌜a⌝.

Invocation

a.x(b$_1$, ... ,b$_n$)   for $n \geq 0$

If ⌜x⌝ is a method field containing a method of n+1 arguments, then the method is invoked by supplying ⌜(a, b$_1$, ... ,b$_n$)⌝ as its arguments, evaluated from left to right. The computed result (or error, or exception) is returned. If ⌜x⌝ is an alias field for ⌜x$_0$⌝ of ⌜a$_0$⌝, then ⌜a$_0$.x$_0$(b$_1$, ... ,b$_n$)⌝ is executed. Invocation fails if ⌜x⌝ is not a field of ⌜a⌝. If the object ⌜a⌝ is serialized, the method executes atomically with respect to other methods of the object.

Updating and Overriding

a.x:=b

If ⌜x⌝ is a value or method field of ⌜a⌝, its contents are replaced by ⌜b⌝, If ⌜x⌝ is an alias field for ⌜x$_0$⌝ of ⌜a$_0$⌝, then ⌜a$_0$.x$_0$:=b⌝ is executed. The result value is ⌜ok⌝. The operation fails if ⌜x⌝ is not a field of ⌜a⌝. The operation fails if it is not self-inflicted and ⌜a⌝ is protected.

Cloning

clone(a$_1$, ... ,a$_n$)   for $n \geq 1$

Provided that all the fields in the ⌜a$_i$⌝ have distinct names, cloning produces an object whose field names are the union of the field names of the ⌜a$_i$⌝, and whose contents are *identical* (section A.2.1) to the contents of the corresponding fields of the ⌜a$_i$⌝. The attributes of the resulting object (protection and serialization) are the same as the attributes of ⌜a$_1$⌝. Cloning fails if one of the ⌜a$_i$⌝ is protected. Cloning is not in general an atomic operation, but it acts atomically on each ⌜a$_i$⌝ that is serialized. The operation fails if it is not self-inflicted on all the ⌜a$_i$⌝'s that are protected.

Delegation

a$_1$.x$_1$:=alias x$_2$ of a$_2$ end

The field ⌜x$_1$⌝ of ⌜a$_1$⌝ is replaced by an alias to the field ⌜x$_2$⌝ of ⌜a$_2$⌝, whether or not ⌜x$_1$⌝ already is aliased. The operation fails if ⌜x$_2$⌝ is not a field of ⌜a$_2$⌝, or if it is not self-inflicted and ⌜a$_1$⌝ is protected.

delegate a$_1$ to a$_2$ end

The fields of ⌜a_1⌝ are replaced by aliases to the similarly named fields of ⌜a_2⌝. This is an atomic operation (even if ⌜a_1⌝ is not serialized): either all or none of the fields of ⌜a_1⌝ are replaced by aliases. The operation fails if ⌜a_2⌝ lacks some of the fields of ⌜a_1⌝, or if it is not self-inflicted and ⌜a_1⌝ is protected.

A.2.7 Protection and Serialization

Every object has two attributes that may or may not be enabled: *protection* and *serialization*. First we need the following definitions; let ⌜*op*(`o`)⌝ be either a select/invoke, update/override, clone, or delegate operation on an object ⌜`o`⌝:

The *current method* of a thread (if it exists) is the last method that was invoked during the thread's execution but has not yet returned.

An object operation ⌜*op*(`o`)⌝ is *self-inflicted* iff ⌜`o`⌝ is identical to the self of the current method (if any).

This definition remains valid under circumstances where threads span multiple sites, and where object identity tests are to be applied to remote objects.

On a *protected* object, all non self-inflicted update/override, cloning, and delegation operations produce errors. Self-inflicted update/override, cloning, and delegation, and all selection/invocation operations are allowed. Protected objects are declared as follows:

```
{protected, ... }
```

A *serialized* object has an associated (implicit) mutex. All non self-inflicted operations acquire the mutex on entry, and release it on completion. Self-inflicted operations do not affect the mutex. Serialized objects are declared as follows:

```
{serialized, ... }
```

A.2.8 Object and Engine Servers

The built-in ⌜`net`⌝ library enables the initial network transmission of objects and engines, by the mediation of a name server. An object can be exported to a name server by saying:

```
net_export("obj", NameServer, o)
```

where ⌜`o`⌝ is the object, ⌜*`NameServer`*⌝ is a text containing the IP address of the machine running the desired name server (⌜`""`⌝ is an abbreviation for the local machine), and the text ⌜`"obj"`⌝ is the registration name for the object. The object is then available through the name server, as long as the site that registered it is alive. Registering under an existing name overrides the previous registration. . The result of this operation is the object ⌜`o`⌝.

Similarly, an engine can be registered with a name server:

```
net_exportEngine("eng", NameServer, arg)
```

where ⌜`arg`⌝ is a value passed to every procedure executed by the engine. The result is ⌜`ok`⌝.

At a separate site (or the same site), an object can be imported:

```
net_import("obj", "tsktsk.pa.dec.com")
```

Now, all object operations can be applied to the resulting remote object.

Similarly, a registered engine can be imported:

```
    net_importEngine("eng", NameServer);
```

The resulting value can be used as a procedure of one argument that, when given a procedure of one argument, returns the result of applying that procedure to the ⌜`arg`⌝ specified in `"exportEngine"`.

Each engine execution takes place in the thread of the client. Hence, sequential calls to an engine from a site execute sequentially. But calls from multiple sites, or from multiple threads within a site, execute concurrently.

The final operation available in the ⌜`net`⌝ library is a net inquiry. It can be applied to objects and engines, and returns a string:

```
    net_who(o)
```

Communication failures raise the exception ⌜`net_failure`⌝.

Certain Obliq built-in values make sense only at the local site, and 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 to the network. In the case of forms [Avrahami, Brooks, Brown 1989], it is possible to transmit a textual form description, and generate the form remotely.

Readers and writers (appendix B.6.11 and B.6.12) can be transmitted over the network; then they operate as efficient network streams. However, their usage is significantly restricted [Birrell, *et al.* 1994]; it is safe to transmit each reader/writer only once away from a site, and from then on to use it only at the receiving site, where it can be retransmitted with the same restrictions.

The alternative of packaging readers/writers within network objects is less efficient, because buffering is then done at the wrong end. However, such packaged readers/writers do not suffer from the usage restrictions above, since they are not transmitted. The restrictions are still in effect on remote cloning of objects containing readers/writers. But this does not interfere with object migration (cloning plus delegation to remote clones), as long as the readers/writers are accessed only through methods, so that no additional transmissions occur.

A.2.9 Processor and File System Enablers

At each site, an *enabler* for the local processor is bound to the predefined, lexically scoped identifier ⌜`processor`⌝. The primitives that start external processes (e.g. Unix processes) require a processor enabler as a parameter. Processor enablers cannot be transmitted.

At each site, an enabler for the local file system is bound to the predefined, lexically scoped identifier ⌜`fileSys`⌝. Moreover, an enabler for a read-only version of the local file system is bound to ⌜`fileSysReader`⌝. The primitives that open files require a file system enabler as a parameter.

File system enablers can be transmitted; multiple file systems can therefore be used at once. Because of lexical scoping, a roaming agent can access the file system of its originating site by referring to ⌜`fileSys`⌝ or ⌜`fileSysReader`⌝ as a free identifier.

Enablers cannot be obtained dynamically, since they are lexically bound. Therefore, roaming agents cannot start local processes, nor access local file systems, unless local enablers are given to them explicitly as parameters.

A.3 Control Structures

In this section we describe the Obliq control structures, including procedures and methods.

A.3.1 Definitions

There are three kinds of definitions binding identifiers to values or locations. They can be used either in a local scope or at the top-level.

$$\texttt{var x}_1\texttt{ = a}_1\texttt{, ..., x}_n\texttt{ = a}_n$$
$$\texttt{let x}_1\texttt{ = a}_1\texttt{, ..., x}_n\texttt{ = a}_n$$
$$\texttt{let rec x}_1\texttt{ = p}_1\texttt{, ..., x}_n\texttt{ = p}_n$$

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 ⌜$\texttt{a}_\texttt{i}$⌝ are all scoped in the context outside the definition. In the third case, the procedures ⌜$\texttt{p}_\texttt{i}$⌝ 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 A.3.3 (sequencing) about local scopes.

A.3.2 Assignment

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 4, a variable can be a network reference.

A.3.3 Sequencing

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

$$\texttt{a}_1\texttt{; ...; a}_n$$

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

A.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. In all cases, arguments are evaluated from left to right.

A.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. The boolean conditions are executed in sequence, and the ⌜then⌝ branch corresponding to the first ⌜true⌝ condition is executed; otherwise the ⌜else⌝ branch is executed (if absent, ⌜ok⌝ is returned).

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 *)

A.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_i⌝ is executed in a scope where ⌜x_i⌝ (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.

A.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 a_1 do a_2 end

foreach x in a_1 map a_2 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.

A.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 similar names [Horning, *et al.* 1993]. The full thread interface is described in appendix B.6.10.

The ⌜mutex⌝ primitive returns a new mutex. The ⌜lock⌝ statement 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 second parameter is the stack size for the thread, in words (0 defaults to a small but non-zero stack size). The ⌜join⌝ primitive waits for the termination of a thread and returns the value of the procedure it executed. The ⌜pause⌝ primitive pauses the current thread for a number of seconds, expressed as a real number.

mutex()

lock a_1 do a_2 end

fork(a_1,a_2)

join(a)

pause(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), then locks the mutex again.

condition()

signal(a)

broadcast(a)

wait(a_1,a_2)

The ⌜watch⌝ statement is specific to serialized objects, and operates on their implicit mutex. Thus, it must occur within a method of a serialized object.

watch a_1 until a_2 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 implicit mutex of the self of the lexically enclosing method.

A.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 text string argument, which is the exception name. Two exceptions are equal if their names are equal text strings. (Hence, an exception can be easily trapped at a site 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, on the left of ⌜`=>`⌝, must be exception values; if an exception is matched, the corresponding branch is executed, otherwise the ⌜`else`⌝ branch is executed. A try-finally statement executes ⌜a_1⌝, and then executes ⌜a_2⌝ no matter whether ⌜a_1⌝ raised an exception; if it did, the exception is raised again.

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

The semantics of try statements with respect to exceptions is the same as in Modula-3. In particular, an exception may propagate across sites, while unwinding the stack of a given thread. See section A.3.10 for their behavior with respect to errors.

A.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 a program. These situations include divide-by-zero, array overrunning, bad operator arguments, and all cases that would produce typechecking errors in typed languages. There are no user-defined errors.

The occurrence of an error indicates a problem that should be fixed by recoding. However, errors are not complete 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 liberally.

Just like exceptions, errors are propagated across sites. Unless something is done, an error in a server caused by a client thread will propagate back to the client, leaving the server unaffected.

A.4 Methodology

A.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 supports a stylized form of comments that are intended to communicate type information, but 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, procedures, and modules. One need write only as much type information as is useful and convenient; type comments have no effect after parsing.

Type comments are used in section B to specify the built-in libraries. Here are examples of the syntax of “types” and their intended meaning:

`Top`, `Ok`, `Bool`, `Char`, `Text`, `Int`, `Real`, `Exception`, `Rd`, `Wr`, `Thread(T)`, `Mutex`, `Condition`, `Process`, `Color`, `Form`	▷ Conventional type and operator names for the built-in types. ⌜`Top`⌝ is the type of all values.
`X`	▷ A user-defined type (any identifier, capitalized by convention).
$\mathtt{X(A_1, \ldots, A_n)}$	▷ A parameterized type, e.g. ⌜`List(Int)`⌝.
$\mathtt{A_1\ op\ A_2}$	▷ An infix parameterized type, e.g. ⌜`Int + Bool`⌝.
`[A]`	▷ The type of arrays of ⌜`A`⌝’s.
`[n*A]`	▷ The type of arrays of ⌜`A`⌝’s of length ⌜`n`⌝ (an integer).
$\mathtt{(A_1, \ldots, A_n)\text{->}A\ !\ exc_1 \ldots exc_m}$	▷ The type of procedures of argument types ⌜$\mathtt{A_i}$⌝ ($n \geq 0$), result type ⌜`A`⌝, and exceptions ⌜$\mathtt{exc_i}$⌝ (where ⌜$\mathtt{!\ exc_1 \ldots exc_n}$⌝ may be omitted).
$\mathtt{(A_1, \ldots, A_n)\text{=>}A\ !\ exc_1 \ldots exc_m}$	▷ The type of methods of argument types ⌜$\mathtt{A_i}$⌝ ($n \geq 0$), result type ⌜`A`⌝, and exceptions ⌜$\mathtt{exc_i}$⌝ (where ⌜$\mathtt{!\ exc_1 \ldots exc_n}$⌝ may be omitted). The type of the self argument is not included in ⌜$\mathtt{A_i}$⌝.
$\mathtt{\{x_1{:}A_1, \ldots, x_n{:}A_n\}}$	▷ The type of objects with components named ⌜$\mathtt{x_i}$⌝ of field type or method type ⌜$\mathtt{A_i}$⌝.
$\mathtt{Option\ x_1{:}A_1, \ldots, x_n{:}A_n\ end}$	▷ The type of options with choices named ⌜$\mathtt{x_i}$⌝ of type ⌜$\mathtt{A_i}$⌝.
`Self(X) B{X}`	▷ Where ⌜`B{X}`⌝ is an object type with possible covariant occurrences of ⌜`X`⌝. This construction is used to give a name (⌜`X`⌝) to the type of the methods’ self (e.g. for objects with methods that return self).
`All(X<:A) B{X}`	▷ Where ⌜`B{X}`⌝ is any type with possible occurrences of ⌜`X`⌝. This is the type of values that, for all subtypes ⌜$\mathtt{A_0}$⌝ of ⌜`A`⌝, have type ⌜$\mathtt{B\{A_0\}}$⌝. If ⌜`<:A`⌝ is omitted, it stands for ⌜`<:Top`⌝.
`Some(X<:A) B{X}`	▷ Where ⌜`B{X}`⌝ is any type with possible occurrences of ⌜`X`⌝. This is the type of values that, for some (unspecified) subtypes ⌜$\mathtt{A_0}$⌝ of ⌜`A`⌝, have type ⌜$\mathtt{B\{A_0\}}$⌝. If ⌜`<:A`⌝ is omitted, it stands for ⌜`<:Top`⌝.

For the last two cases, we say that ⌜`A`⌝ is a subtype of ⌜`B`⌝ (⌜`A<:B`⌝) if every value of type ⌜`A`⌝ is also a value of type ⌜`B`⌝.

Types can be used in the following contexts:

`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. The ⌜`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 ! exc`$_1$` ... exc`$_n$` , b end`

(As opposed to ⌜`proc(x`$_1$`, .., x`$_n$`) b end`⌝.) A commented procedure heading; any of the ⌜`:A`$_i$⌝, ⌜`:A`⌝, and ⌜`! exc`$_1$ ... `exc`$_n$⌝ (the exceptions) may be omitted. The last ⌜`,`⌝ is required only if the result type and/or the exception list is present. Similarly for methods.

`{x`$_1$`:A`$_1$`=>a`$_1$` ..., x`$_n$`:A`$_n$`=>a`$_n$`}`

(As opposed to ⌜`{x`$_1$`=>a`$_1$`, .., x`$_n$`=>a`$_n$`}`⌝.) A commented object; any of the ⌜`:A`$_i$⌝ may be omitted.

`All(X) proc(x:X):X, x end`

The identity function which, for any argument of any type ⌜`T`⌝, returns its argument.

`Some(X) Self(S) {x:X=>0, f:Int=>meth(s:S) s.x+1 end}`

An element of the "abstract type" ⌜`Some(X) {x:X, f:Int}`⌝ with hidden implementation ⌜`X`⌝ = ⌜`Int`⌝. Moreover, ⌜`S`⌝ is used as the type of self.

`module M export type A=Int, x:A, f(x:A,y:A):Bool; ...`

Emphasizing the intended exports of a module, and their types.

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

A.5 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, we have the following pseudo-characters:

a StringChar is either:
- any single character that is not an Illegal character or one of ⌜'⌝, ⌜"⌝, ⌜\⌝.
- two characters `\c`, where `c` is any character that is not Illegal.
- four characters `\xxx`, where `xxx` is an octal number less than 256.

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

an EndOfFile is a fictitious character following the last character in a file or stream.

The following *lexemes* are formed from characters and pseudo-characters:

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 ⌜"⌝.
Nat	a sequence of Digits
Int	a Nat, possibly preceded by a single minus sign ⌜~⌝.
Real	an Int, and either: an ⌜e⌝ and an Int; or a ⌜.⌝, an optional Nat, and optionally an ⌜e⌝ and an Int.
Delimiter	a single Delimiter character.
EndOfFile	a single EndOfFile pseudo-character.

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

Space lexemes do not produce tokens.
Char, String, Int, Real, Delimiter, and EndOfFile lexemes are also tokens.
AlphaNum and Symbol lexemes are Identifier tokens,
except when they have been declared to be *keywords* (see A.6),
in which case they are Keyword tokens.

A.6 Syntax

The grammar shown below is LL(1) and non-left-recursive. It is adapted, with minor editing, from the Obliq metaparser input. See A.5 for the definition of lexical tokens.

Terminals are in double quotes ⌜"⌝. Non-terminals are declared by ⌜::=⌝, followed by a grammar. Grammars have the following structure:

{ g_1 .. g_n }	is a (left-to-right) choice of grammars g_i.
[g_1 .. g_n]	is a sequence of grammars g_i.
(g_1 * g_2)	is g_1 followed by zero or more g_2's, associating to the left.
(g)	is grouping.

ide	recognizes an Identifier token
name	recognizes an Identifier or Keyword token
char	recognizes a Char token

string	recognizes a String token
int	recognizes an Int token
real	recognizes a Real token
EOF	recognizes an EndOfFile token
"..."	where ⌜...⌝ is a Delimiter token: recognizes that Delimiter token
"~..."	where ⌜...⌝ is an Identifier token: recognizes that Identifier token
"..."	where ⌜...⌝ is an Identifier token: declares that identifier to be a keyword and recognizes that Keyword token

The Obliq top-level syntax is an open-ended sequence of the non-terminal “phrase”:

```
phrase ::=
  { ";"
    [ "~help"  { name string [] } { name string [] } ";" ]
    [ "~flag"  { name string [] } { name string [] } ";" ]
    [ typDecl ";" ]
    [ term { [ "!" { int [] } ] [] } ";" ]
    [ "load"  { name string } ";"  ]
    [ "import" name ";" ]
    [ "module" name { [ "for" name ] [] }
      { [ "import" importList  ]  [] } { [ "export" exportList ] [] } ";"  ]
    [ "end" "module" ";" ]
    [ "~establish" name { [ "for" name ] [] } ";" ]       (* reserved *)
    [ "~delete" name ";" ]                                (* reserved *)
    [ "~save" name ";" ]                                  (* reserved *)
    [ "~qualify" ";" ]                                    (* reserved *)
    EOF }

importList ::=
  { [ name  { [ "," importList ] [] } ]  [] }

exportList ::=
  { [ typDecl { [ "," exportList ] [] } ] [ procDecl { [ "," exportList ] [] } ] [] }

typDecl ::=
  [ "type" name { typParams [] } "=" typ ]

typ ::=
  {  [ "(" typList ")" { [ "->" typ ] [ "=>" typ ] [] } ]
     [ "Option" typFields "end" ]
     [ "{" typFields "}" ]
     [ "[" { [ int "~*" ] [] } typ "]" ]
     [ "All" "(" name { [ "<:" typ ] [] } ")" typ ]
     [ "Some" "(" name { [ "<:" typ ] [] } ")" typ ]
     [ "Self" "(" name ")" typ ]
     [ name { [ "_" name { typParams [] } ] typParams [] } ] }

typParams ::=
  [ "(" typNameList ")" ]

typNameList ::=
  { [ name { [ "," typList ] [] } ] [] }

typList ::=
  { [ typ { [ "," typList ] [] } ] [] }

typFields ::=
  { [ name ":" typ { [ "," typFields ] [] } ] [] }

typSpec ::=
  { [ ":" typ ] [] }

typResSpec ::=
  { [ ":" typ { [ "!" excList ] [] } ] [ "!" excList ] }
```

```
excList ::=
  { [ name { [ "_" name ] } excList ] [] }

procDecl ::=
  { [ { "All" "Some" } "(" name { typBound [] } ")" procDecl ]
    [ name { [ ":" typ ] [ "(" ideList ")" typResSpec ] [] } ] }

termBinding ::=
  { [ ide typSpec "=" term { [ "," termBinding ] [] } ] [] }

termSeq ::=
  [ term { [ ";" { termSeq [] } ] [] } ]

termSeqOpt ::=
  { termSeq []  }

term ::=
  ( termBase *
    { [ "(" termList ")" ]
      [ "_" name { [ "(" termList ")" ]  [] } ]
      [ "." name { [ ":=" termOrAlias ] [ "(" termList ")" ] [] } ]
      [ ":=" term ]
      [ "[" term
        { [ "]" { [ ":=" term ] [] } ]
          [ "for" term "]" { [ ":=" term ] [] } ] } ]
      [ ide term ]
      [ "andif" term ]
      [ "orif" term ]  } )

termBase ::=
  {
    [ "~-" term ]
    ide
    { "ok" "true" "false" char string int real }
    [ "[" termList "]" ]
    [ "{" { [ "protected" { "," [] } ] [] } { [ "serialized" { "," [] } ] [] }
      termObjFields "}" ]
    [ "option" name typSpec "=>" termSeqOpt "end" ]
    [ "clone" "(" termList ")" ]
    [ "delegate" termSeq "to" termSeq "end" ]
    [ "proc" "(" ideList ")" { [ typResSpec "," ] [] } termSeqOpt "end" ]
    [ "meth" "(" ideList ")" { [ typResSpec "," ] [] } termSeqOpt "end" ]
    [ "(" termSeqOpt ")" ]
    [ "let" { [ "rec" termBinding ] termBinding } ]
    [ "var" { [ "rec" termBinding ] termBinding } ]
    [ "if" termSeq "then" termSeqOpt termElsif ]
    [ "case" termSeq "of" termCaseList ]
    [ "loop" termSeqOpt "end" ]
    "exit"
    [ "for" ide typSpec "=" term "to" term "do" termSeqOpt "end" ]
    [ "foreach" ide typSpec "in" term { [ "do" termSeqOpt ] [ "map" termSeqOpt ] } "end" ]
    [ "exception" "(" term ")" ]
    [ "raise" "(" term ")" ]
    [ "try" termSeqOpt
      { [ "except" termTryList "end" ]
        [ "else" termSeqOpt "end" ]
        [ "finally" termSeqOpt "end" ] } ]
    [ "lock" termSeq "do" termSeqOpt "end" ]
    [ "watch" termSeq "until" termSeq "end" ]
    [ "All" "(" name { [ "<:" typ ] [] } ")" term ]
    [ "Some" "(" name { [ "<:" typ ] [] } ")" term ]
    [ "Self" "(" name ")" term ]
  }

termOrAlias ::=
  { term [ "alias" ide "of" termSeq "end" ] }
```

```
termObjFields ::=
  { [ name typSpec "=>" termOrAlias { [ "," termObjFields ] [] } ] [] }

termElsif ::=
  { [ "end" ]
    [ "else" termSeqOpt "end" ]
    [ "elsif" termSeq "then" termSeqOpt termElsif ] }

termList ::=
  { [ term { [ "," termList ] [] } ] [] }

ideList ::=
  { [ ide typSpec { [ "," ideList ] [] } ] [] }

termCaseListEnd ::=
  { "end" [ "else" termSeqOpt "end" ] }

termCaseList ::=
  { termCaseListEnd
    [ name
      { [ "(" ide typSpec ")" "=>" termSeqOpt { [ "," termCaseList ] termCaseListEnd } ]
        [ "=>" termSeqOpt { [ "," termCaseList ] termCaseListEnd } ] } ] }

termTryList ::=
  { [ "else" termSeqOpt ]
    [ term "=>" termSeqOpt { [ "," termTryList ] [ "else" termSeqOpt ] [] } ]
    [] }
```

B. System Reference

This section contains information about running Obliq executables, handling source files, and using built-in libraries.

B.1 The Executables

The ⌜`obliq`⌝ Unix shell command is a script that runs one of several versions of Obliq linked with different Modula-3 libraries, providing different built-in Obliq libraries. Network capabilities are supported in all versions of Obliq.

Here are the executables currently provided, along with the supported built-in libraries:

`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 linking 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`⌝.

B.2 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.

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";  (* text concatenation *)
"this is a single text"

- 3 is 4;                        (* identity test *)
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, see ⌜`help flags;`⌝.

B.3 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 use an explicit text string containing a file name (relative to the current search path), or an explicit file path:

```
- load "Foo.obl";
- load "/udir/luca/Foo.obl";
```

The search path for loading is set by the environment variable OBLIQPATH, and can be changed via the ⌜`sys`⌝ built-in library (see B.6.1, or ⌜`help sys;`⌝).

At startup time, the Obliq system looks for a file called ⌜`.obliq`⌝ in the user's HOME directory, and loads it if it finds it.

B.4 Modules

Obliq modules are used for: (1) organizing, loading, and reloading collections of definitions, and (2) for turning collections of definitions into libraries, so that qualified names can be used for the defined identifiers. Modules neither hide nor create scopes, except for turning identifiers into qualified identifiers when a module is closed.

An Obliq source file should normally contain a single module. But, in general, multiple modules can be stored in the same file, and modules can also be entered directly at the top-level. Both the top-level and the source files may contain definitions that are not grouped into modules.

Modules can be used to record source file dependencies: when loading a module, the dependent modules are automatically loaded while avoiding duplicated loading. Modules also help keep the top-level consistent when reloading, for example after a bug fix. 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 before the module is reloaded.

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*. Closing ⌜`Foo`⌝ means erasing its definitions from the current scope, and adding back a library named ⌜`Foo`⌝ containing those definitions. Hence, any top-level identifier ⌜`x`⌝ declared within ⌜`Foo`⌝ is accessible as ⌜`Foo_x`⌝ after closing. (The syntax ⌜`m_x`⌝ is the same as for the built-in libraries.)

If ⌜`end module;`⌝ is omitted, the module is said to be *open*: its identifiers are accessible simply as ⌜`x`⌝. Closed modules should be the norm, but open modules are useful for importing definitions into the top level, and for allowing pervasive unqualified definitions.

If a module ⌜`Foo`⌝ relies on definitions stored in other program files (which should similarly start with ⌜`module`⌝ lines), then ⌜`Foo`⌝ can begin with the line:

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

The way the imported definitions are used within ⌜`Foo`⌝ depends on whether the imported modules are open or closed.

When issuing the top-level command ⌜`load Foo;`⌝, the module declaration above guarantees two properties: (1) if the modules `Foo2` and `Foo3` have not been loaded already, they are loaded before `Foo` is loaded; (2) 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`. This roll-back affects only the top-level definition environment: it does not undo state changes.

The form ⌜`module Foo for L` ...⌝ indicates a collection of definitions named ⌜`Foo`⌝ that generates a library named ⌜`L`⌝ (instead of the default ⌜`Foo`⌝) when the module is closed. Module names are unique at the top level (any repetition triggers roll-back), but library names can be repeated. When multiple modules generate the same library ⌜`L`⌝, their definitions are merged, with the latter ones taking precedence. Using this mechanism, it is possible to add definitions to built-in libraries, for example by ⌜`module text2 for text;`⌝.

B.5 The Network Objects Daemon

A name server must be running before ⌜`net_export`⌝ and similar operations can work. Obliq uses the name server provided with Modula-3 Network Objects [Birrell, *et al.* 1994], it can be started by the Unix command ⌜`netobjd`⌝.

To start a name server on your machine every time the Obliq interpreter starts, put the following line in the ⌜`.obliq`⌝ file in your home directory (make sure the ⌜`netobjd`⌝ path is appropriate):

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

The server process exits if it finds another copy of itself already running.

Note that objects and engines exported via the ⌜net⌝ interface are not inherent security risks, even when they blindly execute client code. The operating system and file system of a server site are not necessarily available (see section A.2.9); lexical scoping prevents any unauthorized access.

B.6 Built-in Libraries

In this appendix we list the Obliq built-in libraries, many of which are entry points into popular Modula-3 libraries [Horning, *et al.* 1993]. We use an informal typing notation in the specification of the operations, including a specification of the exceptions that may be raised (see section A.4.1). Many operations raise errors as well, but these are not made explicit.

We use the type comments of section A.4.1; all the exception conditions are documented, but the more obvious error conditions are not. We often provide informal English descriptions of the operations. For details of some operations one should look at the specification of the respective Modula-3 interfaces [Horning, *et al.* 1993].

The ⌜sys⌝ library is special: it contains entry points into the implementation of Obliq and its computing environment.

B.6.1 Sys

`All(T)`**`sys_copy`**`(x: T): T ! net_failure`	▷ (also ⌜copy(x)⌝) Make a local copy of a value, including most distributed values.
`All(T)`**`sys_print`**`(x: T, depth: Int): Ok`	▷ 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: Text, next: Text): Ok`	▷ Set the interactive prompts (defaults: first=⌜"- "⌝, next=⌜" "⌝). (Only available on-line.)
`sys_address``: Text`	▷ The current machine's network address.
`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_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 program parameter (indexed from 0).
`sys_callFailure``: Exception`	▷ Can be raised by Modula-3 code during a sys_call.
`Some(T)Some(U)`**`sys_call`**`(name: Text, args: [T]): U ! sys_callFailure`	▷ Call a pre-registered Modula-3 procedure.

B.6.2 Bool

`true``: Bool`	▷ The constant true.
`false``: Bool`	▷ The constant false.
`All(T)All(U)`**`bool_is`**`(x: T, y: U): Bool`	▷ (also infix ⌜is⌝) Identity predicate: value equality for Ok, Bool, Int, Real, Char, Text, Exception; pointer equality otherwise.
`All(T)All(U)`**`bool_isnot`**`(x: T, y: U): Bool`	▷ (also infix ⌜isnot⌝) Negation of ⌜is⌝.
`bool_not``(b: Bool): Bool`	▷ (also ⌜not(b)⌝)
`bool_and``(b1: Bool, b2: Bool): Bool`	▷ (also infix ⌜and⌝)
`bool_or``(b1: Bool, b2: Bool): Bool`	▷ (also infix ⌜or⌝)

B.6.3 Int

n: Int	▷ Positive integer constants.
~n: Int	▷ Negative integer constants.
int_minus(n: Int): Int	▷ Integer negation.
int_+(n1: Int, n2: Int): Int	▷ Integer addition.
int_-(n1: Int, n2: Int): Int	▷ Integer difference.
int_*(n1: Int, n2: Int): Int	▷ Integer multiplication.
int_/(n1: Int, n2: Int): Int	▷ Integer division.
int_%(n1: Int, n2: Int): Int	▷ (also infix ⌜%⌝) Integer modulo.
int_<(n1: Int, n2: Int): Bool	▷ Integer less-than predicate.
int_>(n1: Int, n2: Int): Bool	▷ Integer greater-than predicate.
int_<=(n1: Int, n2: Int): Bool	▷ Integer no-greater-than predicate.
int_>=(n1: Int, n2: Int): Bool	▷ Integer no-less-than predicate.

B.6.4 Real

n.m: Int	▷ Positive real constants; m is optional.
~n.m: Int	▷ Negative real constants; m is optional.
real_minus(n: Real): Real	▷ (also ⌜-n⌝) Real negation.
real_minus(n: Int): Int	▷ (also ⌜-n⌝) Overloaded integer negation.
real_+(n1: Real, n2: Real): Real	▷ (also infix ⌜+⌝) Real addition.
real_+(n1: Int, n2: Int): Int	▷ (also infix ⌜+⌝) Overloaded integer addition.
real_-(n1: Real, n2: Real): Real	▷ (also infix ⌜-⌝) Real difference.
real_-(n1: Int, n2: Int): Int	▷ (also infix ⌜-⌝) Overloaded integer difference.
real_*(n1: Real, n2: Real): Real	▷ (also infix ⌜*⌝) Real multiplication.
real_*(n1: Int, n2: Int): Int	▷ (also infix ⌜*⌝) Overloaded integer multiplication.
real_/(n1: Real, n2: Real): Real	▷ (also infix ⌜/⌝) Real division.
real_/(n1: Int, n2: Int): Int	▷ (also infix ⌜/⌝) Overloaded integer division.
real_<(n1: Real, n2: Real): Bool	▷ (also infix ⌜<⌝) Real less-than predicate
real_<(n1: Int, n2: Int): Bool	▷ (also infix ⌜<⌝) Overloaded integer less-than predicate
real_>(n1: Real, n2: Real): Bool	▷ (also infix ⌜>⌝) Real greater-than predicate
real_>(n1: Int, n2: Int): Bool	▷ (also infix ⌜>⌝) Overloaded integer greater-than predicate
real_<=(n1: Real, n2: Real): Bool	▷ (also infix ⌜<=⌝) Real no-greater-than predicate
real_<=(n1: Int, n2: Int): Bool	▷ (also infix ⌜<=⌝) Overloaded integer no-greater-than pred.
real_>=(n1: Real, n2: Real): Bool	▷ (also infix ⌜>=⌝) Real no-less-than predicate.
real_>=(n1: Int, n2: Int): Bool	▷ (also infix ⌜>=⌝) Overloaded integer no-less-than pred.
real_float(n: Int): Real	▷ (also ⌜float(n)⌝) Integer-to-real conversion.
real_float(n: Real): Real	▷ (also ⌜float(n)⌝) Overloaded; identity on reals.
real_round(n: Real): Int	▷ (also ⌜round(n)⌝) Real-to-integer rounding.
real_round(n: Int): Int	▷ (also ⌜round(n)⌝) Overloaded; identity on integers.
real_floor(n: Real): Int	▷ Greatest integers no greater than n.
real_floor(n: Int): Int	▷ Overloaded; identity on integers.
real_ceiling(n: Real): Int	▷ Least integers no less than n.
real_ceiling(n: Int): Int	▷ Overloaded; identity on integers.

B.6.5 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: Real, m: Real): Real	▷ sqrt((n*n)+(m*m)).
math_pow(n: Real, 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: Real, m: Real): Real`	▷ Arc tangent of n/m in radians.

B.6.6 Ascii

`c``: Char`	▷ 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`⌝.

B.6.7 Text

`t``: Text`	▷ 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: Text, 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: Int, size: Int): Text`	The subtext beginning at ⌜`start`⌝, and of size ⌜`size`⌝ (if it exists).
`text_&``(t1: Text, t2: Text): Text`	▷ (also infix ⌜`&`⌝) The concatenation of two texts.
`text_precedes``(t1: Text, 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 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 formatting character is replaced by an escape sequence.
`text_explode``(seps: Text, t: Text): [Text]`	▷ Splits 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: [Text]): Text ! net_failure`	▷ 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_toInt``(t: Text): Int`	▷ Convert a text to an integer (see also fmt_).
`text_fromInt``(n: Int): Text`	▷ Convert an integer to a text (see also lex_).
`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: Text, new: Text, t: Text): Text`	▷ Replace all occurrences of ⌜`old`⌝ by ⌜`new`⌝ in ⌜`t`⌝, as found by iterating ⌜`findFirst`⌝.

B.6.8 Array

[e$_1$, ..., e$_n$]: [T]	▷ (for e1...en: T)
All(T)**array_new**(size: Int, init: T): [T]	▷ An array of size ⌜size⌝, all filled with ⌜init⌝.
All(T)**array_gen**(size: Int, proc: (Int)->T): [T]	▷ An array of size ⌜size⌝, filled with ⌜proc(i)⌝ for ⌜i⌝ between ⌜0⌝ and ⌜size-1⌝.
All(T)**array_#**(a: [T]): Int ! net_failure	▷ (also ⌜#(a)⌝) Size of an array.
All(T)**array_get**(a: [T], i: Int): T ! net_failure	▷ (also ⌜a[i]⌝) The i-th element (if it exists), zero-based.
All(T)**array_set**(a: [T], i: Int, b: T): Ok ! net_failure	▷ (also ⌜a[i]:=b⌝) Update the i-th element (if it exists).
All(T)**array_sub**(a: [T], i: Int, n: Int): [T] ! net_failure	▷ (also ⌜a[i for n]⌝) A new array, filled with the elements of ⌜a⌝ beginning at ⌜i⌝, and of size ⌜n⌝ (if it exists).
All(T)**array_upd**(a: [T], i: Int, n: Int, b: [T]): Ok ! net_failure	▷ (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]⌝.
All(T)**array_@**(a1: [T], a2: [T]): [T] ! net_failure	▷ (also infix ⌜@⌝) A new array, filled with the concatenation of the elements of ⌜a1⌝ and ⌜a2⌝.

B.6.9 Net

net_failure: Exception	
All(T)**net_who**(o: T): Text ! net_failure thread_alerted	▷ Return a text indicating where a network object or engine is registered, or the empty text if the argument is an object that has not been registered with a name server.
All(T<:{})**net_export**(name: Text, server: Text, o: T): T ! net_failure thread_alerted	▷ Export an object under name ⌜name⌝, to the name server at IP address ⌜server⌝. The empty text denotes the local IP address.
Some(T<:{})**net_import**(name: Text, server: Text): T ! net_failure thread_alerted	▷ Import the object of name ⌜name⌝, from the name server at IP address ⌜server⌝. The empty text denotes the local IP address.
All(T)**net_exportEngine**(name: Text, server: Text, arg: T): Ok ! net_failure thread_alerted	▷ Export an engine under name ⌜name⌝, to the name server at IP address ⌜server⌝. The empty text denotes the local IP address. The ⌜arg⌝ is given as an argument to all procedures received by the engine to execute.
Some(T)All(U)**net_importEngine**(name: Text, server: Text): ((T)->U)->U ! net_failure thread_alerted	▷ Import the object of name ⌜name⌝, from the name server at IP address ⌜server⌝. The empty text denotes the local IP address.

B.6.10 Thread

thread_mutex(): Mutex	▷ (also ⌜mutex()⌝) A new mutex.
thread_condition(): Condition	▷ (also ⌜condition()⌝) A new condition.
Some(T)**thread_self**(): Thread(T)	▷ The current thread.
All(T)**thread_fork**(f: ()->T, stackSize: Int): Thread(T)	▷ (also ⌜fork(f,n)⌝) Fork a new thread executing f. If stackSize is zero, a small default size is used.
All(T)**thread_join**(th: Thread(T)): T	▷ (also ⌜join(th)⌝) Wait for a thread to complete, and return the result of its procedure.

thread_wait(mx: Mutex, cd: Condition): Ok	▷ (also ⌜wait(mx,cd)⌝) Wait on a mutex and a condition.
thread_acquire(mx: Mutex): Ok	▷ Acquire a mutex (use lock ... end instead).
thread_release(mx: Mutex): Ok	▷ Release a mutex (use lock ... end instead)
thread_broadcast(cd: Condition): Ok	▷ (also ⌜broadcast(cd)⌝) Wake-up to all threads waiting on a condition.
thread_signal(cd: Condition): Ok	▷ (also ⌜signal(cd)⌝) Wake-up at least one thread waiting on a condition.
thread_pause(r: Real): Ok	▷ (also ⌜pause(r)⌝) Pause the current thread for r seconds.
All(T)**thread_lock**(m: Mutex, body: ()->T): T	▷ Execute under a locked mutex (use lock ... end instead).
thread_alerted: Exception	▷ (See the threads spec.)
All(T)**thread_alert**(t: Thread(T)): Ok	▷ (See the threads spec.)
thread_testAlert(): Bool	▷ (See the threads spec.)
thread_alertWait(mx: Mutex, cd: Condition): Ok ! thread_alerted	▷ (See the threads spec.)
All(T)**thread_alertJoin**(th: Thread(T)): Ok ! thread_alerted	▷ (See the threads spec.)
thread_alertPause(r: Real): Ok ! thread_alerted	▷ (See the threads spec.)

B.6.11 Rd

rd_failure: Exception	
rd_eofFailure: Exception	
rd_new(t: Text): Rd	▷ A reader on a text (a Modula-3 TextRd).
rd_stdin: Rd	▷ The standard input (the Modula-3 Stdio.Stdin).
rd_open(fs: FileSystem, t: Text): Rd ! rd_failure	▷ Given a file system and a file name, returns a reader on a file (a Modula-3 FileRd, open for read). The local file system is available through the predefined lexically scoped identifier ⌜fileSys⌝. Moreover, ⌜fileSysReader⌝ is a read-only version of the local file system.
rd_getChar(r: Rd): Char ! rd_failure rd_eofFailure thread_alerted	▷ Get the next character from a reader.
rd_eof(r: Rd): Bool ! rd_failure thread_alerted	▷ Test for the end-of-stream on a reader.
rd_unGetChar(r: Rd): Ok	▷ Put the last character obtained by getChar back into the reader (unfortunately, it may crash if misused!).
rd_charsReady(r: Rd): Int ! rd_failure	▷ The number of characters that can be read without blocking.
rd_getText(r: Rd, n: Int): Text ! rd_failure thread_alerted	▷ Read the next n characters, or at most n on end-of-file.
rd_getLine(r: Rd): Text ! rd_failure rd_eofFailure thread_alerted	▷ Read the next line and return it without including the end-of-line character.
rd_index(r: Rd): Int	▷ The current reader position.
rd_length(r: Rd): Int ! rd_failure thread_alerted	▷ Length of a reader (including read part).
rd_seek(r: Rd, n: Int): Ok ! rd_failure thread_alerted	▷ Reposition a reader.
rd_close(r: Rd): Ok ! rd_failure thread_alerted	▷ Close a reader.
rd_intermittent(r: Rd): Bool	▷ Whether the reader is stream-like (not file-like).
rd_seekable(r: Rd): Bool	▷ Whether the reader can be repositioned.
rd_closed(r: Rd): Bool	▷ Whether the reader is closed.

B.6.12 Wr

wr_failure: Exception	
wr_new(): Wr	▷ A writer to a text (a Modula-3 TextWr).
wr_toText(w: Wr): Text	▷ Emptying a writer to a text..
wr_stdout: Wr	▷ The standard output (the Modula-3 Stdio.Stdout).

wr_stderr: Wr ▷ The standard error (the Modula-3 Stdio.Stderr).

wr_open(fs: FileSystem, t: Text): Wr ! wr_failure
▷ Given a file system and a file name, returns a writer to the beginning of a file (a Modula-3 FileWr, open for write). The local file system is available through the predefined lexically scoped identifier ⌜fileSys⌝.

wr_openAppend(fs: FileSystem, t: Text): Wr ! wr_failure
▷ Given a file system and a file name, returns a writer to the end of file (a Modula-3 FileWr, open for append). The local file system is available through the predefined lexically scoped identifier ⌜fileSys⌝.

wr_putChar(w: Wr, c: Char): Ok ! wr_failure thread_alerted
▷ Put a character to a writer .

wr_putText(w: Wr, t: Text): Ok ! wr_failure thread_alerted
▷ Put a text to a writer .

wr_flush(w: Wr): Ok ! wr_failure thread_alerted
▷ Flush a writer: all buffered writes to their final destination.

wr_index(w: Wr): Int ▷ The current writer position

wr_length(w: Wr): Int ! wr_failure thread_alerted
▷ Length of a writer.

wr_seek(w: Wr, n: Int): Ok ! wr_failure thread_alerted
▷ Reposition a writer.

wr_close(w: Wr): Ok ! wr_failure thread_alerted
▷ Close a writer.

wr_buffered(w: Wr): Bool ▷ Whether the writer is buffered.

wr_seekable(w: Wr): Bool ▷ Whether the writer can be repositioned.

wr_closed(w: Wr): Bool ▷ Whether the writer is closed.

B.6.13 Pickle

pickle_failure: Exception

All(T)**pickle_write**(w: Wr, v: T): Ok ! pickle_failure wr_failure thread_alerted
▷ Copy a value to a writer, similarly to sys_copy.

Some(T)**pickle_read**(r: Rd): T ! pickle_failure rd_failure rd_eofFailure thread_alerted
▷ Copy a value from a reader, similarly to sys_copy.

B.6.14 Lex

lex_failure: Exception

lex_scan(r: Rd, t: Text): Text ! rd_failure thread_alerted
▷ Read from r the longest prefix formed of characters listed in t, and return it.

lex_skip(r: Rd, t: Text): Ok ! rd_failure thread_alerted
▷ Read from r the longest prefix formed of characters listed in t, and discard it.

lex_match(r: Rd, t: Text): Ok ! lex_failure rd_failure thread_alerted
▷ Read from r the string t and discard it; raise failure if not found.

lex_bool(r: Rd): Bool ! lex_failure rd_failure thread_alerted
▷ Skip blanks, and attempt to read a boolean from r.

lex_int(r: Rd): Int ! lex_failure rd_failure thread_alerted
▷ Skip blanks, and attempt to read an integer from r.

lex_real(r: Rd): Real ! lex_failure rd_failure thread_alerted
▷ Skip blanks, and attempt to read a real from r.

B.6.15 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(t: Text, length: Int): 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	▷ Convert a real to its printable form.

B.6.16 Process

process_new(pr: Processor, nameAndArgs: [Text], mergeOut: Bool): Process	▷ Create a process from a processor and the given process name and arguments. The local processor is available as the lexically scoped identifier ⌜processor⌝. If mergeOut is true, use a single pipe for stdout and stderr.
process_in(p: Process): Wr	▷ The stdin pipe of a process.
process_out(p: Process): Rd	▷ The stdout pipe of a process.
process_err(p: Process): Rd	▷ The stderr pipe of a process.
process_wait(p: Process): Int	▷ Wait for the process to exit, close all its pipes, and return the exit code.
process_filter(pr: Processor, nameAndArgs: [Text], input: Text): Text ! net_failure	▷ Create a process from a processor and the given process name and arguments. The local processor is available as the lexically scoped identifier ⌜processor⌝. The stderr output is merged stdout. Usage: feed the input to its stdin pipe and close it; read all the output from its stdout pipe and close it; return the output.

B.6.17 Color

color_named(name: Text): Color	▷ Get a color from its name (see the ColorName Modula-3 interface).
color_rgb(r: Real, g: Real b: Real): Color	▷ Get a color from rgb (each 0.0 .. 1.0).
color_hsv(h: Real, s: Real v: Real): Color	▷ Get a color from hsv (each 0.0 .. 1.0).
color_r(c: Color): Real	▷ The red color component.
color_g(c: Color): Real	▷ The green color component.
color_b(c: Color): Real	▷ The blue color component.
color_h(c: Color): Real	▷ The hue color component.
color_s(c: Color): Real	▷ The saturation color component.
color_v(c: Color): Real	▷ The value color component.
color_brightness(c: Color): Real	▷ The total brightness (0.0 .. 1.0).

B.6.18 Form

form_failure: Exception	
form_new(t: Text): Form ! form_failure	▷ Read a form description from a text.
form_fromFile(file: Text): Form ! form_failure thread_alerted	▷ Read a form description from a file.
form_attach(fv: Form, name: Text, f: (Form)->Ok): Ok ! form_failure	▷ 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, property: Text): Bool ! form_failure	▷ Get the boolean value of the named property of the named interactor. (Do not confuse with form_getBoolean.)
form_putBool(fv: Form, name: Text, property: Text, b: Bool): Ok ! form_failure	▷ Set the boolean value of the named property of the named interactor. (Do not confuse with form_putBoolean.)
form_getInt(fv: Form, name: Text, property: Text): Int ! form_failure	▷ 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: Text, property: Text, n: Int): Ok ! form_failure
▷ 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: Text, property: Text): Text ! form_failure
▷ 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: Text, property: Text, t: Text, append: Bool): Ok ! form_failure
▷ Set the text value of the named property of the named interactor. If property is the empty text, set the ⌜"value"⌝ property.

form_getBoolean(fv: Form, name: Text): Bool ! form_failure
▷ Get the boolean value of the named boolean-choice interactor.

form_putBoolean(fv: Form, name: Text, b: Bool): Ok ! form_failure
▷ Set the boolean value of the named boolean-choice interactor.

form_getChoice(fv: Form, radioName: Text): Text ! form_failure
▷ Get the choice value of the named radio interactor.

form_putChoice(fv: Form, radioName: Text, choiceName: Text): Ok ! form_failure
▷ Set the choice value of the named radio interactor.

form_getReactivity(fv: Form, name: Text): Text ! form_failure
▷ Get the reactivity of the named interactor. It can be ⌜"active"⌝, ⌜"passive"⌝, ⌜"dormant"⌝, or ⌜"vanished"⌝.

form_putReactivity(fv: Form, name: Text, r: Text): Ok ! form_failure
▷ Set the reactivity of the named interactor. It can be ⌜"active"⌝, ⌜"passive"⌝, ⌜"dormant"⌝, or ⌜"vanished"⌝.

form_popUp(fv: Form, name: Text): Ok ! form_failure
▷ Pop up the named interactor.

form_popDown(fv: Form, name: Text): Ok ! form_failure
▷ Pop down the named interactor.

form_insert(fv: Form, parent: Text, t: Text, n: Int): Ok ! form_failure
▷ Insert the form described by t as child n of parent.

form_move(fv: Form, parent: Text, child: Text, toChild: Text, before: Bool): Ok ! form_failure
▷ Move child before or after toChild of parent; after ⌜""⌝ means first, before⌜""⌝ means last.

form_delete(fv: Form, parent: Text, child: Text): Ok ! form_failure
▷ Delete the named child of parent.

form_deleteRange(fv: Form, parent: Text, n: Int, count: Int): Ok ! form_failure
▷ Delete count children of parent, from child n.

form_takeFocus(fv: Form, name: Text, select: Bool): Ok ! form_failure
▷ Make the named interactor acquire the keyboard focus, and optionally select its entire text contents.

form_show(fv: Form): Ok ! form_failure
▷ Show a window containing the form on the default display.

form_showAt(fv: Form, at: Text, title: Text): Ok ! form_failure
▷ Show a window containing the form on a display. For an X display: at=⌜"*machineName*(‘:’|‘::’)*num*(‘’|‘.’*num*)"⌝; at=⌜""⌝ is the default display. The title is shown in the window header.

form_hide(fv: Form): Ok ! form_failure
▷ Hide the window containing the form.

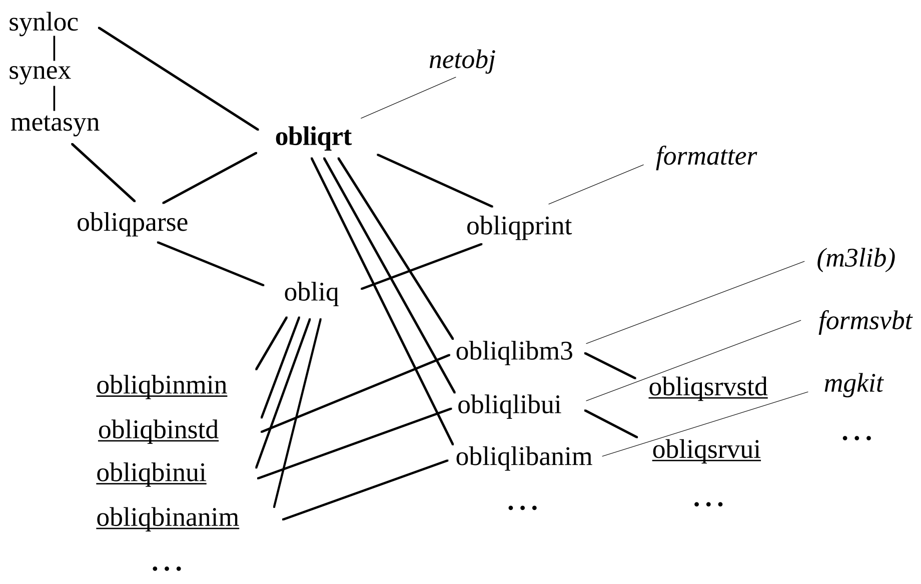
C. Programming Reference

In this section we provide information useful to programmers who want to call Obliq from Modula-3, or vice versa.

C.1 The Package Hierarchy

One of our goals is that Obliq should be easily embeddable in Modula-3 applications. Obliq adds only a small size overhead to typical Modula-3 applications, but we still want to minimize this overhead. To this end, the Obliq implementation is partitioned into several packages, with a Modula-3 library in each package, so that each application can link only the appropriate libraries. Another advantage of this organization, is that we can generate minimal Obliq interpreters that can act as (relatively) small network servers.

Here is the package structure. Each node is a package (a collection of interfaces), which uses the connected packages above it. The nodes in italic represent packages external to the Obliq implementation.



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⌝ 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⌝ and ⌜obliqprint⌝.

The ⌜obliq⌝ package brings together everything needed to build stand-alone Obliq interpreters. This package can be linked with various library packages to produce various flavors of Obliq interpreters.

Each underlined package contains a short Main program and a binary for an interpreter (⌜-bin-⌝) or a server (⌜-srv-⌝).

Modula-3 programmers can extend the hierarchy along the dotted lines.

C.2 The Interfaces

The main client interface is ⌜obliqrt/src/Obliq.i3⌝, which refers to ⌜obliqrt/src/ObTree.i3⌝ (the parse trees) and ⌜obliqrt/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 need not be linked into an application, since an application may access evaluated objects and closures over the network. The main interface to the parser is ⌜obliqparse/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/src/ObliqPrinter.i3⌝.

C.3 The Libraries

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

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

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

C.4 Embedding Obliq in an Application

The appropriate client interfaces are ⌜obliqrt/src/Obliq.i3⌝, ⌜obliqparse/src/ObliqParser.i3⌝, and ⌜obliqprint/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⌝.

The Obliq evaluator takes as arguments a syntax tree, and an environment. The environment, mapping identifiers to Obliq values, is particularly important. By manipulating the environment, one can submit values to Obliq for evaluation, and can recover the results of an evaluation.

C.5 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 C.6.

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

One must then link the Modula-3 code implementing ⌜`foo`⌝ with Obliq, either in an application (section C.4) or in a custom interpreter (section C.7).

C.6 Extending Obliq with new Packages

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

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

C.7 Building a Customized Obliq Interpreter

A new package, created as described in section C.6, can be embedded into a customized Obliq interpreter. Follow the example given by ⌜obliqbinstd/src/Main.m3⌝: this is the 20-line program that builds the standard Obliq interpreter. The other ⌜obliqbin.../src/Main.m3⌝ files contain other versions of the interpreter.

References

[Abadi, Cardelli 1994] M. Abadi and L. Cardelli. **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag.

[Apple 1993] Apple, **AppleScript Language Guide**. Addison Wesley.

[Avrahami, Brooks, Brown 1989] G. Avrahami, K.P. Brooks, and M.H. Brown, **A two-view approach to constructing user interfaces**. *Computer Graphics* **23**(3), 137-146.

[Bal, Kaashoek, Tanenbaum 1992] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum, **Orca: a language for parallel programming of distributed systems**. *IEEE Transactions on Software Engineering* **18**(3), 190-205.

[Bharat, Brown 1994] K. Bharat and M.H. Brown. **Building distributed applications by direct manipulation**. Digital Equipment Corporation, Systems Research Center. To appear.

[Birrell 1991] A.D. Birrell, **An introduction to programming with threads**. In *Systems Programming with Modula-3, Chapter 4,* G. Nelson, ed. Prentice Hall.

[Birrell, *et al.* 1993a] A.D. Birrell, D. Evers, G. Nelson, S. Owicki, and E. Wobber. **Distributed garbage collection for network objects**. Report 116. Digital Equipment Corporation, Systems Research Center.

[Birrell, *et al.* 1993b] A.D. Birrell, G. Nelson, S. Owicki, and E. Wobber. **Network objects**. *Proc. 14th Symposium on Operating Systems Principles.*

[Birrell, *et al.* 1994] A.D. Birrell, G. Nelson, S. Owicki, and E. Wobber. **Network objects**. Report 115. Digital Equipment Corporation, Systems Research Center.

[Brewer, Waldspurger 1992] E.A. Brewer and C.A. Waldspurger. **Preventing recursion deadlock in concurrent object-oriented systems**. *Proc. 1992 International Parallel Processing Symposium, Beverly Hills, California. (Also, Report MIT/LCS/TR-526.).*

[Brockschmidt 1994] K. Brockschmidt, **Inside OLE2**. Microsoft Press.

[Brown 1994] M.H. Brown. **Report on the 1993 SRC algorithm animation festival**. Digital Equipment Corporation, Systems Research Center. To appear.

[Brown, Meehan 1994] M.H. Brown and J.R. Meehan. **The FormsVBT Reference Manual**. Digital Equipment Corporation, Systems Research Center. To appear.

[Forté 1994] Forté. **TOOL reference manual**. Forté, Inc.

[Horning, *et al.* 1993] J. Horning, B. Kalsow, P. McJones, and G. Nelson. **Some useful Modula-3 interfaces**. Report 113. Digital Equipment Corporation, Systems Research Center.

[Mansfield 1994] R. Mansfield, **Visual Basic for Applications**. Ventana Press.

[Milner, Tofte, Harper 1989] R. Milner, M. Tofte, and R. Harper, **The definition of Standard ML**. MIT Press.

[Najork, Brown 1994] M. Najork and M.H. Brown. **A library for visualizing combinatorial structures**. *Proc. Visualization'94*. To appear.

[Nelson 1991] G. Nelson, ed. **Systems programming with Modula-3**. Prentice Hall.

[Ousterhout 1994] J.K. Ousterhout, **Tcl and the Tk toolkit**. Addison-Wesley.

[Reppy 1991] Reppy. **A higher-order concurrent language**. *Proc. SIGPLAN'91 Conference on Programming Language Design and Implementation*. ACM Press.

[Thomsen, *et al.* 1993] B. Thomsen, L. Leth, S. Prasad, T.-M. Kuo, A. Kramer, F. Knabe, and A. Giacalone. **Facile Antigua Release Programming Guide**. ECRC-93-20. European Computer-Industry Research Centre.

[Ungar, Smith 1987] D. Ungar and R.B. Smith. **Self: the power of simplicity**. *Proc. OOPSLA'87*. ACM SIGPLAN Notices 2(12).

[White 1994] J.E. White. **Telescript technology: the foundation for the electronic marketplace**. White Paper. General Magic, Inc.

Index