The
Oaklisp Language Manual

December 22, 2014

Barak A. Pearlmutter
Hamilton Institute
National University of Ireland Maynooth
Co. Kildare
Ireland
barak+oaklisp@cs.nuim.ie

Kevin J. Lang
Yahoo! Research
langk@yahoo-inc.com

Contents

Chapter 1

Introduction

This is the introduction to the original Oaklisp proposal which we wrote in January 1985. Although the core language hasn't changed since then, some of the peripheral ideas in the proposal have been modified or abandoned.

One of the most interesting language ideas to emerge from the 1970's was the object-oriented programming model. Although this model has been incorporated to some extent in a number of recent Lisps, these implementations have not had the generality and power that characterize a true object-based system like Smalltalk. The most significant trend in the contemporary Lisp world is the move toward lexical scoping, which was initiated by Steele and Sussman with their Scheme papers and continued most faithfully by the designers of T.

The major goal of Oaklisp was to combine the ideas of Smalltalk and Scheme in a simple but expressive language that inherits their exemplary properties of modularity and consistency. Unlike T, which adds an object-based capability to Scheme by constructing objects out of closures, Oaklisp builds a lexically scoped Lisp system on top of a general message-passing system that allows for full inheritance of methods from multiple superclasses.

The first design choice for Oaklisp was the extent to which we should push the object-based model. We agreed with the designers of ADA that the packaging of state and procedures into new types can be a significant modularity tool for users of the language. We also agreed with the designers of the Lisp Machine that single-user Lisp systems should be open, with no clear line between system and user code. Therefore, both to allow the system implementors to use powerful user-level tools and to allow users to easily manipulate the system, we decided that absolutely everything should be a full-fledged object in Oaklisp. There is no reason why user-defined types should be any different from the types used to build the system. Oaklisp stands in marked contrast to other object-oriented Lisp systems which have magic data types that are not part of the user-level type hierarchy.[1] Our current Oaklisp implementation takes this idea to such an extreme that there are no magic objects anywhere in the system, no matter how deep you go. This meant not only that the vast majority of the system could be written in Lisp, but that the construction of the debugger and garbage collector was greatly simplified.

As a corollary to the previous decision, we decided that all computation should be performed

[1] For example, ZetaLisp flavors are not themselves instances of flavors.

by methods that are invoked after a search up the type hierarchy. Functions can be thought of as methods attached to the top of the hierarchy, since they are methods that can perform an operation on any type. This leads to the interesting result that after a function has been defined, a new method can be added to take over that operation for a special case.

The power of the method-invocation model of computation is derived from the generality of the inheritance mechanism.[2] A simple type-tree would have provided Oaklisp with the ability to define shadowable system-wide defaults for `print` and so forth. However, we felt that the mixin concept of flavors was such a valuable tool for factoring object functionality that inheritance from multiple supertypes was essential. This idea of inheriting from several mixins, each of which knows how to do something and encapsulates its own state, led to the following inheritance rule: a new type inherits all of the methods of its supertypes, but methods for the new type cannot refer to instance variables from the supertypes (even though those variables do exist in the new composite object.) This does not cause problems, because when operations for the supertypes are passed to the object, the methods which handle them *can* reference the appropriate instance variables. Since the names of instance variables are never inherited, conflicts cannot occur between names in the various supertypes.

This treatment of instance variable names was also motivated by our decision to follow Scheme and make Oaklisp lexically scoped. Oaklisp not only benefits from the conceptual correctness which results from being able to close methods at compile time, but takes full advantage of tail-recursion and the lack of search associated with variable references. Once again, we decided to carry a principle to its extreme, and say that *all* variable references must be resolved at compile time, which results in both a simpler compiler and faster execution of code. Although this decision sounds intolerable for users, it actually represents a shift of functionality from the compiler to the error-handling system and user-interface, both of which we decided to make unusually powerful in our Oaklisp implementation.

Another principle which we borrowed from Scheme is anonymity. The lack of coupling between names and objects gives the system a degree of modularity and flexibility that would otherwise be difficult to achieve. For example, if a type is redefined, old instances of that type will still have pointers to the old type descriptor. Operations on both kinds of object will be handled by the correct methods, and when the last instance of the old type goes away, the old type descriptor will also be garbage collected.

The portion of Oaklisp that has been described so far can be considered its kernel. The portion that follows can mostly be implemented as methods at the user level. It is interesting to note that the dynamic variables and mutable binding contours which are described below can be built on top of a bare lexical Lisp kernel. However, the Oaklisp kernel is not a usable system, since we intentionally stripped it down knowing that the lost facilities would be replaced at a higher level.

The first addition is a mutable binding contour facility based the locale structures of T.[3] Oaklisp locales are objects that accept messages which install and look up names. Locales can have multiple superiors which are recursively searched if a name can't be found. Unlike T locales, Oaklisp locales are not associated with textual binding contours that can interact with `let`'s and

[2] It is primarily the lack of inheritance that weakens the T object facility.

[3] In place of locales we could have implemented a simple top-level binding environment.

generate ambiguities. Moreover, a reference to an undefined name creates an error, which means that forward references to uninstalled names are impossible.

The second addition is a dynamic scoping facility that knows how to deal with `catch` and `throw`. The new dynamic variables are entirely separate from the static variables, and are always textually distinguishible from static variables to avoid confusion. Dynamic variables use deep-binding in our implementation so that they will behave correctly when there is more than one process. The implementation of dynamic variables is an issue since we decided to follow the Lisp Machine and implement light-weight processes that share the same address space to expedite data sharing and fast context switching.

Our final design decision was also influenced by the Lisp Machine. Error handling in Oaklisp is designed to take maximum advantage of the type inheritance mechanism that is built into the language. A complete hierarchy exists of all the types of system errors. When an error occurs, an instance of that error type is created, and a message is sent to the error object asking for a handler to take control. The default message causes the debugger to be invoked. However, each process has some dynamic state which can be modified with the `condition-bind` construct to cause a different handler to be invoked when a particular error occurs at a particular time. This mechanism brings the full power of the language to bear on the problem of resolving errors, and is the reason that we felt we could make the language itself so strict with respect to variable references. Since our implementation of Oaklisp runs on the Lisp Machine and the Macintosh, it was no problem to delegate authority for reporting and resolving unbound variable references to the user-interface, which uses menus and dialog boxes to determine the user's intentions. If the user sets a switch that indicates that he wants unbound names to be automatically installed in the innermost locale, then the interface merely creates an error-handler to perform that function, and the user is not bothered again.

Chapter 2

Types and Objects

Oaklisp is an object-oriented language which is organized around the concept of *type*. The type of an object determines its behavior when operations are performed on it. To permit the modular specification of types with complex behaviors, a type is allowed to have multiple supertypes. There is no distinction in Oaklisp between predefined system types and user-defined types.

A type specifies the behavior of an object by providing methods that are used to perform operations on that object. Because methods are inherited from supertypes, a subtype only needs to supply those methods which are required to distinguish itself from the more general types. A method defined for a given type pre-empts any inherited methods for the same operation.

Instance variables are the mechanism for keeping state in objects. Every object possesses a data structure where the values of its instance variables are stored. Although each object contains storage for all of the instance variables required by its type and supertypes, methods for a given type can only refer to instance variables defined in that type. In particular, methods cannot refer to instance variables that are defined in supertypes.

It is possible to think of Oaklisp in terms of messages that are being passed to objects, rather than in terms of operations that are being performed on objects. The latter view was chosen because it is more consistent with Lisp syntax and semantics.

2.1 Fundamental Types

There are two important relations in the Oaklisp type system: *is-a* and *subtype*. An object is related to its type by the relation *is-a*, and a type is related to its supertypes by the relation *subtype*. Each of these relations defines a tree structure which includes all of the objects in the system.

The most fundamental types in the system are `type` and `object`. They are distinguished by their position at the top of the *is-a* and *subtype* hierarchies, and by their circular definitions.

`type` *Type*

This type is the top of the *is-a* hierarchy. It is the type of types, so new types are created by instantiating it.

`object` *Type*

This type is the top of the *subtype* hierarchy, and has no supertype. Every other type is a subtype of `object`, so default methods for operations such as `print` are defined for `object`.

2.2 Operations on Objects

The following operations are defined for all objects. Because they determine the semantics of the language, they cannot be redefined or shadowed.

`(get-type` *`object`*`)` *Operation*

Returns the type of `object`.

`(eq?` *`object object`*`)` *Predicate*

Determines object identity. Two objects may look and act the same, but still fail the `eq?` test. In particular, numbers are not guaranteed to be unique. Symbols *are* interned, though.

2.3 Operations on Types

Types are distinguished from other objects by the fact that they can perform the `make` operation, which is the mechanism for generating new objects.

`(make` *`type`*`)` *Operation*

Returns a new instance of `type`.

The instance variables of an object returned by `make` are all bound to some unspecified value. Usually new objects need to be initialized in some other way, which can be accomplished by performing an operation on them immediately after they are made. By convention, this operation is `initialize`.

`(initialize` *`object`*`)` *Operation*

Returns `object`.

This method for `initialize` is clearly a no-op. When a type requires special initialization, it should shadow this default.

2.4 Defining New Types

Since types are objects, new ones are created by sending a `make` message to the appropriate type object, which in this case is `type`.

`(make type` *`ivars supertypes`*`)` *Operation*

Returns a new type-object with the supertypes and instance variables specified by the argument lists.

At run-time, methods are chosen by performing a left-to-right depth-first search on the supertype list.[1] Instances of the new type will contain a block of instance variables for each of the ancestor types, although duplicate types in the ancestor tree are eliminated.[2]

2.5 Type Predicates

The implicit type checking performed by the method invocation mechanism of Oaklisp reduces the need to call explicit type predicates. Furthermore, the two predicates defined in this section are sufficiently general to replace all of the ordinary Lisp type predicates such as `null?` and `number?`. A few of these have been retained to make the environment more familiar.

`(is-a?` *`object type`*`)` *Predicate*

Determines whether `object` is an instance of *type* or one of its subtypes. `(is-a?` *`object`* `object)` is always true.

`(subtype?` *`type1 type2`*`)` *Predicate*

Determines whether *type1* is a subtype of *type2*. As you would expect, `subtype?` is transitive. Since each type is a subtype of itself, `subtype?` defines a partial ordering of all the types in the system.

2.6 Constants

Some objects have external representations that are not self-evaluating expressions. `quote` allows the inclusion of such objects as constants in code.

`(quote` *`object`*`)` *Special Form*

Returns *object* without evaluating it.

2.7 Standard Truth Values

The standard truth values of Oaklisp are represented by the objects bound to the following variables.

`t` *Global Variable*

The value of this is `#t`. Any non-false value will do just as well for the purpose of logical tests.

`#f` *Global Variable*

This is the false value, the only object recognized by logical tests as denoting falsehood.

`nil` *Global Variable*

[1] Of course, Oaklisp implementations are free to use more efficient mechanisms that have the same effect.

[2] This aspect of the language is in flux, and should not be relied upon by users.

The value of this is the empty list, written `()`. Notice that `nil` itself is just a variable, so `(eq? nil 'nil)` is false.

Note: currently `()` is the same as `#f`, the object used to represent falsehood. In the future it is possible that these two notions, emptiness and falsehood, will be disconfabulated. Programs should be written in such a way that if `#f` and `()` were not the same object, they would still work.

2.8 Coercion

Some types are *coercable*, meaning that there is an operation associated with that type that allows objects to be coerced to that type. To create a coercable type, one instantiates `coercable-type` rather than `type`.

`(coercer` *`coercable-type`*`)` *Locatable Operation*

This returns the coercer of a type. For example, to coerce a list into a string one uses `(coercer string)`, as in `((coercer string) '(#\f #\o #\o))` ⇒ `"foo"`. The reader will read `frog` preceded by a control-y character as `(coercer frog)`; this was motivated by the fact that control-y prints as → on both Macintosh™ and Symbolics computers, giving coercion a pleasant syntax, `(→string '(#\f #\o #\o))` ⇒ `"foo"`.

`coercable-type` *Type*

This is a subtype of `type` with has the added functionality of responding to the `coercer` message by returning its coercion operation. By default, `(is-a?` *`foo bar`*`)` implies that `((coercer` *`bar`*`)` *`foo`*`)` ⇒ *`foo`*

2.9 Mixing Types

Frequently, type hierarchies become so rich that they threaten to overwhelm users with a plethora of possible combinations of mixins. The combinatorial explosion of the number of possible concocted types seems intrinsic to the style of programming involving multiple functionally orthogonal mixins. Above a certain level of complexity, finding a type with certain known characteristics can become difficult. Programmers are left wondering "Has a type based on *foo* with *bar, baz* and *zonk* mixed in been created, if so what's its name, and if not what should I name it and where should I define it?"

Oaklisp's *mixin managers* take care of this problem. When one needs "the type based on *foo* with *bar, baz* and *zonk* mixed in," one asks a mixin manager for it. If such a type has already been created, it is returned; if not, the mixin manager creates an appropriate new type, caches it, and returns it. This eliminates the burden of remembering which types have been concocted and what they are named.

`(mix-types` *`mixin-manager type-list`*`)` *Operation*

This returns a composite type whose supertypes are *type-list*. *Mixin-manager* checks its cache, and if the requested type is not found it creates a type with `(make type '() type-list)`, caches it, and returns it.

`mixin-manager` *Type*

Instances of this cache composite types, acting as a sort of composite type library.

The Oaklisp operation type hierarchy is quite elaborite, containing a large number of functionally orthogonal mixins, and therefore the Oaklisp internals make heavy use of the mxin manager facility when dealing with operations. For example, the following definition for + is drawn from deep within the bowels of Oaklisp.

```
(define-constant-instance +
  (mix-types oc-mixer
             (list foldable-mixin open-coded-mixin operation)))
```

Chapter 3

Methods and Scoping

In Chapter 2, the concept of *type* was discussed. The assertion was made that operation methods lie at the heart of the typing system, because they determine the behavior of objects. This chapter describes the mechanism for defining methods.

3.1 Methods

A table of methods is maintained in the descriptor of every type. At run-time, these tables are searched to find the methods which are used to handle operations on objects. The only mechanism for manipulating method tables is the following side-effecting special form.

`(add-method (`*operation* `[(`*type* `.` *ivar-list*`)] .` *arg-list*`).` *body*`)` ***Special Form***

Adds a method for *operation* to the method table of *type*. If a method for *operation* already exists, it is replaced. The value returned by `add-method` is *operation*.

The body of the form is surrounded by an implicit `block`. The arguments to the method are specified by *arg-list*. Since the first argument is always the object handling the message, a useful convention is to call it `self`. Instance variables of *type* can be referenced in the body if they are declared in *ivar-list*. Instance variables of supertypes may not be referenced in any case. Naming conflicts between instance variables and arguments are resolved by the rule that the variables in *arg-list* shadow instance variables that have the same names. Oaklisp closes methods over free variable references at compile-time, thereby solving the upward funarg problem and allowing procedures to share state in a controlled manner.

3.2 Scoping

Oaklisp is a lexically scoped language in which all variable references are resolved at compile-time. When a variable reference is encountered, the compiler searches outwards from that point through the nested lexical binding contours until it finds a declaration for the variable.[1] We have already

[1] If a declaration isn't found, the compiler proceeds to look for the variable in the appropriate locale. See Chapter 5.

seen one mechanism for introducing new lexical contours: the argument list of the `add-method` special form. Oaklisp provides several other forms which can be used to define local variables and procedures.

`(let ((`var_1 val_1`)...`var_n val_n`). body)` *Special Form*

Evaluates *body* in an environment where the n variables are bound to the n values. The value returned is that of *body*.

`(let* ((`var_1 val_1`)...`var_n val_n`). body)` *Special Form*

This form is similar to `let`. The difference is that `let` performs the bindings simultaneously whereas `let*` performs the bindings sequentially so that each value expression can refer to the preceding variables.

`(labels ((`var_1 val_1`)...(`var_n val_n`)). body)` *Special Form*

`labels` differs from `let` in that the value expressions are evaluated in a binding environment in which all of the variables are already defined. This facilitates the definition of mutually recursive procedures.

3.3 Functional Syntax

Sometimes it is convenient to adopt a more conventional Lisp viewpoint while designing programs. This viewpoint considers functions to be the primary programming abstraction, with objects downgraded to the status of data which is passed around between functions. The key to this programming style is the ability to write functions which can accept arguments of any type.

Oaklisp readily accommodates the functional programming style, since methods can be defined for the type `object`, which is the supertype of all other types. In fact, if the type specifier is omitted in an `add-method` form, the type `object` is assumed. Thus, `(add-method (cons-1 x) (cons x 1))` defines a method that is valid for any type. To give the language a more familiar appearance when this programming style is used, the following macros are also provided.

`(lambda` *arg-list* `.` *body*`)` *Macro*

≡ `(add-method ((make operation)` *arg-list*`)` *body*`)`

`(define (`*variable* `.` *arg-list*`)` *body*`)` *Macro*

≡ `(define` *variable* `(lambda` *arg-list* `.` *body*`))`

3.4 Dispatching to Supertypes

Sometimes a method doesn't want to override the inherited method completely, but rather wishes only to modify or extend its behaviour. For instance, imagine that the type `dog` has a method so that the `notice-stranger` operation causes it to run around, jump up and down, bark, and return the amount of time wasted. Say that `stupid-dog` is a subtype of `dog` defined by

`(define-instance stupid-dog type '() (list dog))`, and that we want stupid dogs to behave just like regular dogs in response to a `see-stranger` message, except that they do it twice. This could be accomplished without the duplication of code by dispatching to the supertype twice, as in the following code fragment.

```
(add-method (see-stranger (stupid-dog) self stranger)
  (+ (↑super dog see-stranger self stranger)
     (↑super dog see-stranger self stranger)))
```

`(↑super` *`type operation self . args`*`)` *Operation*

This is just like `(`*`operation self . args`*`)` except that the method search begins at *type* rather than at the type of *self*. It is required that *type* be an immediate supertype of the type that the method this call appears in is added to, although our current implementation does not yet enforce this restriction. `↑super` is analogous to the Smalltalk-80 mechanism of the same name, except that due to Oaklisp's multiple inheritance it is necessary for the programmer to explicitly state which supertype is to be dispatched to.

3.5 Rest Args

When a method is defined with a parameter list that is improper (*i.e.* dotted) the method is permitted to receive extra values in addition to its regular parameters at run time. These values are associated with the pseudo variable name that appears after the dot, which will henceforth be called the rest name. Unlike a real variable name, a rest name can't be evaluated and can only be referred to in two places: at the end of a function call that uses dotted syntax (which signifies that the extra values should be passed on to the function being called), and in a `rest-length` form, which is the mechanism for finding out how many rest args a method has been passed.

`(rest-length` *`rest-name`*`)` *Special Form*

Yields the number of extra values that were received by the method in which `rest-name` is declared.

Rest args can never be accessed directly, but must be passed tail recursively to other functions. In fact, a function is not permitted to return without disposing of its rest args. Usually a function that takes a variable number of arguments will recurse on itself or on a helper function, consuming its arguments one by one until they are all gone, at which point the function is free to return.

The following functions have been provided to make it easier to write a function definition that satisfies all of the rules for rest args.

`(consume-args` *`val . args`*`)` *Operation*

Returns *val* after consuming *args*.

`(listify-args` *`op . args`*`)` *Operation*

Calls *op* on a list consisting of the values of *args*.

A call to `listify-args` can be used as the body of a method definition as a means of trivially satisfying the rest arg rules. When using this technique, *op* is a lambda that performs all of the computation for the method. The rest args of the method are wrapped up in a list that is passed in as the lambda's one parameter, and the regular parameters and instance variables of the method are available inside the lambda because of lexical scoping.

Chapter 4

Side Effects

The treatment of side effects in Oaklisp is modelled on that of T. The salient feature of this approach is the use of reversible access procedures to perform side effects on composite data structures and anonymous storage cells.

4.1 Assignment

Side effects on variables and objects are performed with the `set!` special form, which combines the functionality of the `setq` and `setf` forms found in other Lisps.

`(set!` *`location new-value`*`)` *Special Form*

Changes the value of *location* to *new-value*, which is then returned as the value of the expression.

If *location* is a symbol, then it is interpreted as a variable name. The variable must have been previously defined in some lexical binding contour or locale.

If *location* is a list, then it is interpreted as a reference to a settable access operation. For example, `(set! (car foo) 'bar)` means the same thing as `(rplaca foo 'bar)` in Common Lisp.

`(setter` *`operation`*`)` *Locatable Operation*

Takes a settable access operation and returns the corresponding alteration operation.

4.2 Locatives

`locative` is an Oaklisp type that is similar to the pointer types found in procedural languages such as C or Pascal. Locatives are created and dereferenced by the following constructs.

`(make-locative` *`location`*`)` *Special Form*

Returns a locative that points to *location*, which must be a variable or a list with the form of a call on a locatable access operation.

`(locater` *`operation`*`)` *Locatable Operation*

Takes a locatable access operation and returns the corresponding locative-making operation.

`(contents` *`locative`*`)` *Locatable Operation*

Returns the contents of the location which is referenced by *locative*. Since `contents` is a settable operation, side effects can be performed on locations through locatives. For example, `(set! (contents (make-locative (car` *`foo`*`))) '`*`bar`*`)` has the same effect as `(set! (car` *`foo`*`) '`*`bar`*`)`.

4.3 Operation Types

Since operations are objects, they are classified into types according to the operations which can be performed on them. The types discussed here can generate side-effecting operations from access operations.

`operation` *Type*

This is the generic operation type that is a component of all other operation types.

`settable-operation` *Type*

An access operation is settable if side effects can be performed through it. Settable operations respond to `setter`.

`locatable-operation` *Type*

An access operation is locatable if it retrieves information from a single physical location. Locatable operations respond to `setter` and `locater`.

4.4 Modification Forms

See Chapter 6 of *The T Manual* for a description of the following forms.

`(swap` *`location new-value`*`)` *Special Form*

`(modify` *`location procedure`*`)` *Special Form*

`(modify-location` *`location continuation`*`)` *Special Form*

Chapter 5

Evaluation and Locales

Locales are the namespace structuring mechanism of Oaklisp. Whenever an Oaklisp expression is evaluated, a locale must be provided in order to specify a particular mapping from symbols to macro-expanders and from symbols to storage cells.

5.1 Evaluation

`(eval` *`form locale`*`)` *Operation*

Evaluates *form* relative to *locale*.

Although programmers don't often need to call `eval` directly, every expression typed at the top level is passed in to `eval` to be evaluated relative to the locale specified by the fluid variable `current-locale`. Files may be evaluated using the `load` function.

`(load` *`file-name`* [*`locale`*]`)` *Operation*

Reads all of the forms in the file *file-name* and evaluates them relative to *locale*, which defaults to the value of `(fluid current-locale)` if not specified.

The file compiler can be used to create an assembly language file that has the same effect as an Oaklisp source file.

`(compile-file` *`locale file-name`*`)` *Operation*

Compiles the file *file-name* relative to *locale*, which defaults to the value of `(fluid current-locale)`. A file must be compiled and loaded relative to the same locale in order to guarantee that the program's semantics are preserved.

Oaklisp source files have a default extension of `.oak` while compiled files are given the extension `.oa`. `compile-file` first tries to read the file *file-name*`.oak`, and then looks for *file-name*, while `load` looks first for *file-name*`.oa`, then for *file-name*`.oak`, and finally for *file-name*.

5.2 Installing Names in a Locale

Oaklisp has several forms that can be used to insert global variables and macro definitions into a locale. The target locale isn't explicitly specified by any of these forms, but is implicitly understood to be the locale with respect to which the form is being evaluated. Thus, when a form is typed at the top level, the effect is on `(fluid current-locale)`, and when a file is loaded, the effect is on the locale specified in the call to `load`.

`(define` *var val*`)` *Special Form*

Installs the global variable *var* in the current locale with value *val*.

`(define-constant` *var val*`)` *Special Form*

This form is like `define` except that *var* is marked as frozen in the current locale so that the compiler can be free to substitute the value for references to *var*.

`(define-instance` *var typ* `.` *make-args*`)` *Special Form*

If the contents of *var* isn't of type *typ*, this is the same as `(set!` *var* `(make` *typ* `.` *make-args*`))`. If *var* is already bound to an object of the right type, this form has no effect. *Note:* this language feature is in flux. Currently, it sends an `initialize` message to the object with the new *make-args*.

`(define-syntax` *macro-name expander*`)` *Special Form*

Installs *macro-name* in the current locale. *expander* should be a lambda that is able to translate an example of the macro into a form that has simpler syntax.

As with all Oaklisp forms, the effect of a `define-syntax` form in a file is not felt until run-time when the file is loaded. Since it is often convenient to be able to use a macro in the file in which it is defined, a special mechanism has been provided for defining file-local macros that are in effect at compile time. The following magic forms should be used with care, since they violate the usually absolute dichotomy between compile time and load time.

`(local-syntax` *macro-name expander*`)` *Special Form*

During the compilation of a file in which a `local-syntax` form is contained, the form augments the name space with the macro specified by *macro-name* and *expander*. This form can only appear at top level in a file; and essentially disappears before load time.

`(define-local-syntax` *macro-name expander*`)` *Special Form*

Temporarily augments the compile-time name space with the specified macro, and also installs the macro in the current locale when the file is loaded. This form can only appear at top level in a file.

5.3 Structuring the Namespace

Oaklisp locales are not associated with textual binding contours, nor are they particularly user-friendly objects. They were designed to be a powerful implementation tool, leaving the task of providing a convenient interactive interface to higher-level code.

`(make locale` *`superior-list`*`)` *Operation*
Returns a new locale which inherits names from the locales in *superior-list*. During recursive name lookups, the superiors are searched deapth first in left-to-right order.

5.4 Variables

Locales are essentially mappings from symbols to storage cells. Although locales can be created on-the-fly, their main use is in building the structured top-level environment for global variables. Variable names must be installed in a locale before they can be referenced. Precise control over shadowing and cross-referencing can be achieved using the following settable operations.

`(variable?` *`locale symbol`*`)` *Settable Operation*
Returns a locative to the appropriate storage cell if *symbol* is installed as a variable name, or `#f` otherwise. The search is allowed to proceed to superior locales if necessary.

`(set! (variable?` *`locale symbol`*`)` *`locative`*`)` *Operation*
If *symbol* is not currently defined at any level, then it is installed in *locale*, with the location named by *locative* serving as its value cell. If *symbol* is defined at some level, then its value cell at the highest level[1] is changed to be the location referenced by *locative*.

`(set! (variable?` *`locale symbol`*`)` *`#f`*`)` *Operation*
If *symbol* is defined at some level, then its definition is removed from the highest level. Otherwise an error is generated.

`(variable-here?` *`locale symbol`*`)` *Settable Operation*
Returns a locative to the appropriate storage cell if *symbol* is installed as a variable name, or `#f` otherwise. The search is constrained to *locale* itself.

`(set! (variable-here?` *`locale symbol`*`)` *`locative`*`)` *Operation*
If *symbol* is not currently defined in *locale*, then it is installed, with the location named by *locative* serving as its value cell. If *symbol* is defined in *locale*, then its value cell is changed to be the location referenced by *locative*.

`(set! (variable-here?` *`locale symbol`*`)` *`#f`*`)` *Operation*
If *symbol* is defined in *locale* then its definition is removed. Otherwise an error is generated.

[1] *i.e.* in the nearest locale to the one handling the operation.

5.5 Macros

Macro definitions are also stored in locales. These definitions are stored as a mapping from names to macro expanders. A macro expander is simply a one-argument function that takes an S-expression as its input and returns a transformed S-expression. Macro definitions are installed with the following settable operations, which are entirely analogous to the ones described in Section 5.4.

`(macro?` *`locale symbol`*`)` *Settable Operation*

Returns the appropriate macro expander if *symbol* is installed as a macro name and `#f` otherwise. The search is allowed to proceed to superior locales if necessary.

`(macro-here?` *`locale symbol`*`)` *Settable Operation*

Returns the appropriate macro expander if *symbol* is installed as a macro name, or `#f` otherwise. The search is constrained to *locale* itself.

5.6 Compilation

All evaluation in Oaklisp is performed with respect to some locale. The syntax of the language is determined by the macro tables visible from that locale, and free variable references are likewise resolved using the global variables defined in its name space.

`(frozen?` *`locale symbol`*`)` *Settable Predicate*

Returns `#t` if *symbol* is a frozen variable, otherwise `#f`. The search is allowed to proceed to superior locales if necessary. If *symbol* is not found anywhere, an error occurs.

`(frozen-here?` *`locale symbol`*`)` *Settable Predicate*

Returns `#t` if *symbol* is a frozen variable, otherwise `#f`. The search is constrained to *locale* itself. If *symbol* is not installed as a variable in *locale*, an error occurs.

Chapter 6

Dynamic State

As Steele and Sussman pointed out in *The Art of the Interpreter*, dynamic scoping provides the most natural decomposition of state in certain situations. This chapter describes the Oaklisp facilities for creating and manipulating state that has dynamic extent.

6.1 Fluid Variables

To avoid the problems that arise when fluid variables are integrated with the lexical environment, Oaklisp fluid variables have been placed in a completely separate dynamic environment. Fluid variables don't even look like lexical variables, since they can only be referenced using the `fluid` special form. The mechanism for creating fluid variables is `bind`, which syntactically resembles `let`.

`(bind ((`*fluid* var_1`)` val_1`)...((`*fluid* var_n val_n`). `*body*`)` *Special Form*

Evaluates *body* in a dynamic environment where the n symbols are bound to the n values.

`(fluid `*symbol*`)` *Special Form*

Returns the value of the fluid variable *symbol*. Even though `fluid` is a special form, it is settable, so `(set! (fluid `*symbol*`) `*value*`)` changes the value of the fluid variable *symbol* to *value*. The reader will read `foo` preceded by a control-v character as `(fluid foo)`; this was motivated by the fact that control-v prints as • on both Macintosh™ and Symbolics computers.

6.2 Non-local Exits

Most Lisp dialects include some sort of `catch` facility for performing non-local exits. Oaklisp provides two facilities at varying points on the generality vs. cost spectrum.

`(call-with-current-continuation` *operation*`)` *Operation*
Calls *operation* with one argument, the current continuation. The synonym `call/cc` is provided for those who feel that `call-with-current-continuation` is excessively verbose.

`(catch` *variable* `.` *body*`)` *Special Form*
variable is lexically bound to an escape operation that may be called from anywhere within *body*'s dynamic extent. If *variable* is not called, `catch` yields the value of *body*. This is implemented in such a way that *body* is called tail recursively.

`(native-catch` *variable* `.` *body*`)` *Special Form*
variable is lexically bound to an escape tag that may be thrown from anywhere within *body*'s dynamic extent. If *variable* is not thrown to, `native-catch` yields the value of *body*. This is implemented in such a way that *body* is called tail recursively.

`(throw` *tag value*`)` *Operation*
Causes execution to resume at the point specified by *tag*. This point is always a `native-catch` expression, which immediately yields *value*. Cleanup actions specified with `wind-protect` are performed while the stack is being unwound.

`(wind-protect` *before form after*`)` *Special Form*
≡ `(dynamic-wind (lambda () ` *before*`) (lambda () ` *form*`) (lambda () ` *after*`))`

`(funny-wind-protect` *before abnormal-before form after abnormal-after*`)` *Special Form*
A `wind-protect` evaluates *before*, *form*, and *after*, returning the value of *form*. If *form* is entered or exited abnormally (due to `call/cc` or `catch`) the *before* and *after* forms, respectively, are automatically executed. `funny-wind-protect` is the same except that different guard forms are evaluated depending on whether the dynamic context is entered or exited normally or abnormally.

`(dynamic-wind` *before-op main-op after-op*`)` *Operation*
Calls the operation *before-op*, calls the operation *main-op*, calls the operation *after-op*, and returns the value returned by *main-op*. If *main-op* is exited abnormally, *after-op* is called automatically on the way out. Similarly, if *main-op* is entered abnormally, *before-op* is called automatically on the way in.

6.3 Error Resolution

Note: the error system is not stable, and will probably evolve towards the Common Lisp error system, which has a number of good ideas.

Programs interact with the error system in three ways: they signal various sorts of errors (typically throwing the user into the debugger), they provide restart handlers that the user can invoke (using `ret`) to escape from the debugger, and they provide handlers to be invoked when various types of errors occur.

6.3.1 Signaling Errors

Errors are signalled using the following operations.

`(warning` *`format-string . format-args`*`)` *Operation*

Prints out the message specified by *format-string* and *format-args* and continues execution.

`(error` *`format-string . format-args`*`)` *Operation*

This signals `generic-fatal-error`, which normally has the effect of printing out the error message specified by *format-string* and *format-args* and dumping the user into a subordinate read-eval-print loop.

`(cerror` *`continue-string format-string . format-args`*`)` *Operation*

This signals `generic-proceedable-error`, which normally has the effect of printing the error message specified by *format-string* and *format-args* and dumping the user into a subordinate read-eval-print loop in which there is a restart handler that continues the computation by returning a user specified value from `cerror`. *Continue-string* is the text associated with this handler when it is listed as an option by the subordinate evaluator.

6.3.2 Restart Handlers

There are two special forms that programs can use to define more complex restart handlers than just returning from the call to `cerror`. The simpler of the two is `error-return`, which is similar to `catch` in that it can be forced to return a value before its body has been fully evaluated. This form is used in the definition of `cerror`.

`(error-return` *`string . body`*`)` *Macro*

Evaluates *body* in a dynamic context in which a restart handler is available that can force the form to return. The handler is identified by *string* in the list of choices the debugger presents to the user. If the handler is invoked by calling `ret` with an

argument in addition to the handler number, the `error-return` form returns this value; otherwise it returns `#f`. If no error occurs, `error-return` yields the value of *body*.

The second special form acts just like a `let` unless an error occurs, in which case an error handler is available that re-executes the body of the form after (possibly) rebinding the lexical variables specified at the top of the form.

`(error-restart` *string* `((`var_0 val_0`)...)` `.` *body*`)` *Macro*

Evaluates *body* in a dynamic context in which a restart handler is available that can force the re-evaluation of the body with new values for $var_0 \ldots$. These new values are specified as additional arguments to `ret`. If there are not enough arguments to `ret`, the remaining variables are left at their previous values. The handler is identified by *string* in the list of choices printed by the debugger. If no error occurs, `error-restart` yields the value of *body*.

6.3.3 Error Handlers

Oaklisp uses its type system to govern the resolution of errors. The top-level environment contains a hierarchy of types which characterizes every error that can occur. When an error condition arises, the appropriate type is instantiated, and an error resolution operation is performed on the new object. This operation is performed by a method that deals with the error in a manner consistent with its type.

There are clearly better ways of dealing with some errors than invoking the debugger. A variety of methods have been written to deal with the most common errors. For example, there are `proceed` methods for simple arithmetic traps which substitute a program specified value for that of the failed computation. The use of `proceed` and other error resolution operations is prescribed by the following special form.

`(bind-error-handler ((`err_1 op_1`)...(`err_n op_n`)).` *body*`)` *Macro*

Evaluates *body* in a dynamic environment where the n error types have been associated with the n error resolution operations. When an error occurs, the current list of condition bindings is searched to find an operation to perform. An operation associated with a supertype of the actual error type will be selected if it is encountered on the list. If a suitable operation cannot be found, the default operation `invoke-debugger` is performed.

6.3.4 Operations on Errors

There are a number of operations that can be invoked on error objects in error handlers.

`(report` *error stream*`)` *Operation*

Instructs *error* to print a descriptive message to *stream*.

`(invoke-debugger` *error*`)` *Operation*

This is the default error resolution operation. It is performed on all errors unless it is explicitly overridden.

`(proceed` *`error . values`*`)` *Operation*

Attempts to continue from the error, eg. a file system error would retry the failed operation. The *values* have semantics determined by the precise type of error. For instance, continuing a failed attempt to open a file with a value might instruct the system to try a new filename.

`(remember-context` *`error after-operation`*`)` *Operation*

Instructs *error* to salt away the current continuation and then call *after-operation*, which should never return.

`(invoke-in-error-context` *`error operation`*`)` *Operation*

Invokes *operation* on *error* after moving back to the context of the error if its been salted away.

6.3.5 Error Types

There are a plethora of error types defined in Oaklisp.

`general-error` *Type*

This is the top of the error type hierarchy. An operation defined for `general-error` can be used to resolve any error.

`generic-fatal-error` *Type*

Signaled by `error`.

`(make proceedable-error` *`message`*`)` *Operation*

Uses *message* in composing its report.

`generic-proceedable-error` *Type*

Signaled by `cerror`.

`error-opening` *Type*

Various subtypes of this are signaled when various types of error while opening files occur.

`read-error` *Type*

Subtypes of this are signaled when `read` sees malformed or truncated input.

`unexpected-eof` *Type*

This subtype of `read-error` is signaled when the reader comes to the end of a file unexpectedly.

Work for idle hands: Many types of errors have yet to be implemented. For example, domain errors in arithmetic functions generally call `error` rather than signaling some special variety of error, template mismatch in the `destructure*` macro should signal some special type of error rather than calling `cerror`, etc. Basically, most calls to `error` and `cerror` in system level code should be replaced with `signal`, and appropriate ideosyncratic types of errors should be defined, thereby giving users more precise control over what types of system level errors to handle.

Chapter 7

Control

Nonlocal control constructs like `call/cc` are described in section 6.2.

Since control structures are not a very interesting issue, we followed existing Lisp dialects closely when designing this aspect of Oaklisp. Every control structure in this chapter does just what you would expect.

7.1 Simple Constructs

These forms are compatible with both T [14, chapter 5] and the Scheme standard [13].

`(cond . clauses)` *Special Form*

The *clauses* are run through sequentially until one is selected. Each clause can be of four possible forms. `(`*test* `.` *body*`)` evaluates *body* if *test* is true. `(else .` *body*`)` always evaluates *body*, and if present must be the last clause. `(`*test* `=>` *operation*`)` calls *operation* on the result of *test* if the result of evaluating *test* was not *false*. `(`*test*`)` is equivalent to `(`*test* `=> identity`)`.

`(if test consequent [alternate])` *Special Form*

`(not object)` *Predicate*

`(and . tests)` *Special Form*

`(or . tests)` *Special Form*

`(iterate variable specs . body)` *Special Form*

`(block . body)` *Special Form*

Evaluates the forms of *body* sequentially, returning (tail recursively) the value of the last one.

`(block0` *form* `.` *body*`)` *Special Form*
≡(let ((x *form*)) (block `.` *body*) x)

`(dotimes (`*variable number* [*rform*]`) .` *body*`)` *Special Form*
≡(let ((x (lambda (*variable*) `.` *body*))) (map x (iota *number*)) *rform*)

`(dolist (`*variable list* [*rform*]`) .` *body*`)` *Special Form*
≡(let ((x (lambda (*variable*) `.` *body*))) (map x *list*) *rform*)

`(dolist-count (`*variable list count-var*`) .` *body*`)` *Special Form*
Just like `dolist` except that *count-var* gives the count of the current element in the list, starting at zero.

`(while` *condition* `.` *body*`)` *Special Form*
≡(let ((q (lambda () *test*))(x (lambda () `.` *body*))) (iterate aux () (cond ((*q*) (*x*) (aux)))))

`(unless` *test* `.` *body*`)` *Special Form*
≡(cond ((not *test*) `.` *body*))

`(do ((`*var initial step* `)...)(`*termination-test* `.` *termination-body* `) .` *body*`)` *Special Form*
≡(iterate aux ((*var initial*) ...) (cond (*termination-test* `.` *termination-body*) (else (block `.` *body*) (aux *step* ...))))

7.2 Mapping Constructs

Although these can be used as control constructs, they can also be thought of as ways to manipulate data structures. `map` maps an operation over some sequences generating a sequence of results. `for-each`, which doesn't save the results, is used when the operation is called for effect only. For all of these, the order of evaluation is undefined; the system may apply the operation to the various elements of the sequence in any order it desires.

`(map` *operation* `.` *sequences*`)` *Operation*

`(mapcdr` *operation* `.` *lists*`)` *Operation*
Applies *operation* to successive "cdrs" rather than to elements, and returns a list of the returned values.

`(for-each` *operation* `.` *sequences*`)` *Operation*

`(for-each-cdr` *operation* `.` *lists*`)` *Operation*
Like `mapcdr` but for effect only.

`(map!` *operation* `.` *sequences*`)` *Operation*
Like `map`, except that the retuned values are destructively placed into the successive storage locations of the first *sequence*.

Chapter 8

Sequences

Sequences are manipulated using the `nth` operation, which is settable (and locatable). The sequence heirarchy is shown in figure 8.1.

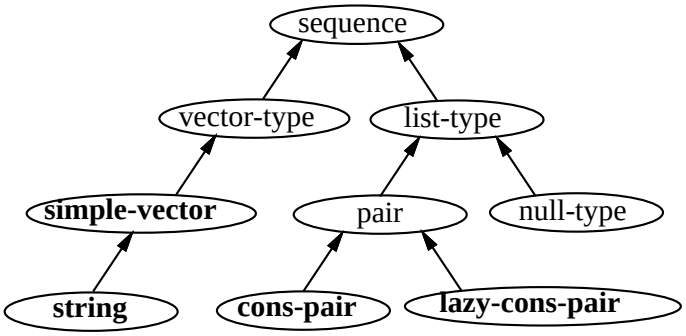

Figure 8.1: The sequence type hierarchy. Abstract types are in plain face and instantiable ones in bold.

8.1 Type Predicates

`(sequence?` *object*`)` *Predicate*

`(vector?` *object*`)` *Predicate*

`(string?` *object*`)` *Predicate*

`(list?` *object*`)` *Predicate*

`(pair?` *object*`)` *Predicate*

(null? *object*) *Predicate*

(atom? *object*) *Predicate*

8.2 Sequence Operations

These operations work on all sequences.

(length *list*) *Operation*

(nth *list n*) *Locatable Operation*

(last *list*) *Locatable Operation*

(tail *list n*) *Locatable Operation*

(copy *sequence*) *Operation*

(append *sequence1 sequence2*) *Operation*

Returns a sequence of the type of *sequence1*. One slight bug is that one may not pass `append` a first argument that's a list and a second that's not. This may be fixed in the future. All other combinations should work correctly.

(append! *sequence1 sequence2*) *Operation*

Most sequences have immutable lengths, and hence are not appropriate arguments to `append!`. The major exception is lists. The same bug is present here as in `append`.

(reverse *sequence*) *Operation*

(reverse! *sequence*) *Operation*

Some mapping operations are also applicable to sequences, and are documented in Section 7.2.

8.3 Vector Constructors

(vector . *objects*) *Operation*

Returns a `simple-vector` containings *objects*.

(make simple-vector *length*) *Operation*

((coercer simple-vector) *sequence*) *Coarcable Type*

8.4 List Constructors

`(list . objects)` *Operation*

`(make list-type length fill-value)` *Operation*

`((coercer list-type) sequence)` *Coarcable Type*

`(cons object1 object2)` *Operation*

`(make lazy-cons-pair car-thunk cdr-thunk)` *Operation*

`(lcons car-form cdr-form)` *Macro*
≡ `(make lazy-cons-pair (lambda () car-form) (lambda () cdr-form))`

8.5 List Accessors

`(car pair)` *Locatable Operation*

`(cdr pair)` *Locatable Operation*

`(c[ad]*r pair)` *Locatable Operation*
Actually these are only provided for up to four `a`'s and `d`'s. If you think you need more, you should probably be defining accessor functions or using `nth` or perhaps `destructure`.

`(last-pair pair)` *Locatable Operation*
Takes successive `cdr`'s of *pair* until it finds a pair whose `cdr` is not a pair, which it returns. `(last-pair '(a b c))` ⇒ `(c)`. `(last-pair '(a b c . d))` ⇒ `(c . d)`.

`(destructure template structure . body)` *Macro*
This is for destructuring lists, and is sort of the inverse of backquote. *Template* is a possibly nested list of variables. These variables are bound to the corresponding values of *structure* while *body* is evaluated. For instance, `(destructure (a (b) . c) x (foo a b c))` ≡ `(let ((a (car x))(b (caadr x))(c (cddr x))) (foo a b c))`
. It is guaranteed that *structure* will be evaluated only once. We note that `destructure` typically generates more efficient code than the corresponding code one might typically write.

If there is a position in *template* that should be ignored, one can place a `#t` there. For convenience and compatiblity with `destructure*`, positions in *template* containing `()`, `#f` and `(quote x)` are also ignored.

`(destructure* template structure . body)` *Macro*

This is just like `destructure` except that an error is signaled if *structure* doesn't precisely match *template*. Positions containing `#f` and `()` are required to match literally. Positions containing `(quote x)` are required to match *x* literally, where *x* is not evaluated. As with `destructure`, positions containing `#t` are ignored.

`destructure*` is particularly useful in macro expanders where it can do much of the syntax checking automatically.

`(destructure** structure (template . body)... [(otherwise . nomatch-bo`
Macro

This is just like `destructure*` except that, when one template does not match, the next in line is considered. If none match than the OTHERWISE one does; if no otherwise clause is present, an error is signaled.

8.6 Lists as Sets

`(mem predicate object list)` *Operation*

Returns the first tail of *list* whose `car` equals *object* according to *predicate*.

`(memq object list)` *Operation*

`(del predicate object list)` *Operation*

`(delq object list)` *Operation*

`(del! predicate object list)` *Operation*

`(delq! object list)` *Operation*

8.7 Lists as Associations

`(ass predicate object list)` *Operation*

`(assq object list)` *Operation*

`(cdr-ass predicate object list)` *Settable Operation*

`(cdr-assq object list)` *Settable Operation*

8.8 Lists as Stacks

(push *location object*) *Macro*

(pop *location*) *Macro*

Chapter 9

Numbers

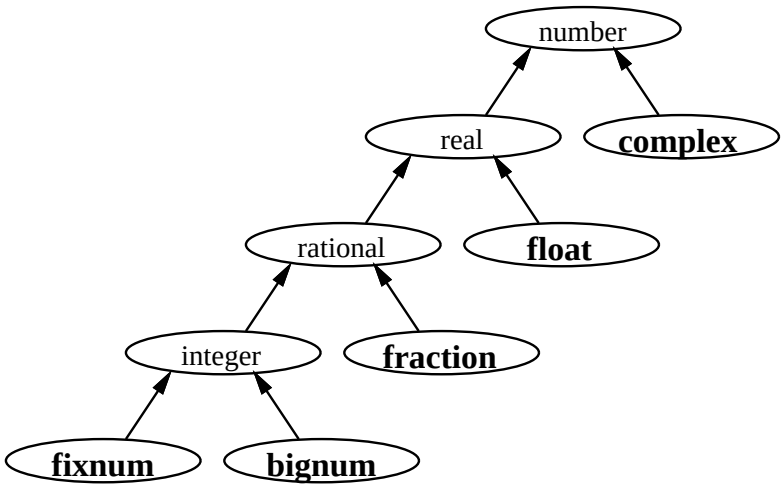

Figure 9.1: The numeric type hierarchy. Abstract types are in plain face and instantiable ones in bold. Floating point numbers are not implemented.

9.1 Arithmetic

`(+ . numbers)` *Operation*

`(1+ n)` *Operation*

`(- n1 n2 . numbers)` *Operation*

`(- n)` *Operation*

(* . *numbers*) *Operation*

(/ *n1 n2*) *Operation*

(quotient *n1 n2*) *Operation*

(modulo *n1 n2*) *Operation*

(abs *n1*) *Operation*

(max *n1 n2*) *Operation*

(min *n1 n2*) *Operation*

(expt *n1 n2*) *Operation*

9.2 Comparison

(= *n1 n2*) *Operation*

(!= *n1 n2*) *Operation*

(< *n1 n2*) *Operation*

(> *n1 n2*) *Operation*

(<= *n1 n2*) *Operation*

(>= *n1 n2*) *Operation*

9.3 Predicates

(zero? *n*) *Predicate*

(negative? *n*) *Predicate*

(positive? *n*) *Predicate*

`(even? n)` *Predicate*

`(odd? n)` *Predicate*

`(factor? n1 n2)` *Predicate*

9.4 Rounding

These operations should work on any subtype of `real`.

`(floor x)` *Operation*

Returns the largest integer less than or equal to *x*.

`(ceiling x)` *Operation*

Returns the smallest integer greater than or equal to *x*.

`(truncate x)` *Operation*

Could be defined `(if (negative? x) (ceiling x) (floor x))`.

`(round x)` *Operation*

Returns nearest integer to *x*. Ties are broken by rounding to an even number.

9.5 Bitwise Logical Operations

These operations are only defined for integers.

`(ash-left i amount)` *Operation*

`(ash-right i amount)` *Operation*

`(rot-left i amount)` *Operation*

`(rot-right i amount)` *Operation*

`(bit-not i)` *Operation*

`(bit-and i1 i2)` *Operation*

`(bit-or i1 i2)` *Operation*

`(bit-nor i1 i2)` *Operation*

(bit-xor *i1 i2*) *Operation*

(bit-nand *i1 i2*) *Operation*

(bit-andca *i1 i2*) *Operation*

(bit-equiv *i1 i2*) *Operation*

9.6 Accessing Components

(numerator *rational*) *Operation*

(denominator *rational*) *Operation*

(real-part *number*) *Operation*

(imag-part *number*) *Operation*

Chapter 10

Input and Output

10.1 Streams and Files

Streams are the tokens through which interaction with the outside world occurs. Although streams are primarily used for reading and writing to files, they have found a number of internal uses.

`stream` *Type*

The supertype of all streams.

`input-stream` *Type*

This is an abstract type. Instantiable subtypes must define methods for the `really-read-char` operation.

`(read-char` *`input-stream`*`)` *Operation*

Return a character, or `the-eof-token` if we've already read the last character in the stream.

`(unread-char` *`input-stream character`*`)` *Operation*

Puts *character* back into *input-stream*. One can only put one character back, and it must be the last character read.

`(peek-char` *`input-stream`*`)` *Operation*

Equivalent to `(let ((c (read-char` *`input-stream`*`))) (unread-char` *`input-stream`* `c) c)`.

`the-eof-token` *Object*

This distinguished object is returned to indicate that one has read past the end of the file.

`output-stream` *Type*

This is an abstract type. Instantiable subtypes must define methods for the `write-char` operation.

`(write-char output-stream character)` *Operation*

`(newline output-stream)` *Operation*

Outputs a carriage return to *output-stream*.

`(freshline output-stream)` *Operation*

Ensures that *output-stream* is at the beginning of a line.

`(flush output-stream)` *Operation*

Flushes any buffered output.

`(interactive? stream)` *Operation*

Returns true if and only if *stream* is connected to the user. This is used to check if an end of file condition on the control stream is really an end of file or if the user just typed control-D.

`(position stream)` *Settable Operation*

Returns the position we are at within *stream*. By setting this, one can get back to a previous position.

`(write-string string output-stream)` *Operation*

Writes the characters of *string* to *stream*.

`(with-open-file (variable filename . options). body)` *Macro*

Binds *variable* to a stream which is connected to the file with the name *filename*. *Options* is not evaluated, and describes how *filename* should be opened. Possible symbols include `in` for input, `out` for output, and `append` for output with position set to the end of the file. The `ugly` option can be added to either `out` or `append` if the user doesn't mind poor formating, as in files meant to be read only by other programs. The opened stream will be closed when the `with-open-file` is exited, even upon abnormal exit. **Note:** the stream is not reopened upon abnormal entry, but this may be changed in future versions of the system.

`(with-input-from-string (variable sequence). body)` *Macro*

Binds *variable* to an input stream whose contents are the characters of *sequence*. Although *sequence* is usually a string, this will work correctly for any sequence type.

`(make string-output-stream )` *Operation*

These save all their output and return it as a string in response to the `(coercer string)` operation.

10.2 Reading

Oaklisp has an industrial strength reader, replete with nonterminating macro characters and descriptive error messages. List syntax is not described below; read some other lisp manual. Our reader is modeled after the Common Lisp reader, so we emphasize differences with the Common Lisp reader below.

`(read` *`input-stream`*`)` *Operation*

Returns a lisp object read from *stream*. This is sensitive to a large number of factors detailed below.

`standard-read-table` *Object*

This holds the read table for usual lisp syntax. The `nth` operation can be used to get and set elements of read tables, which are indexed by characters. Potential entries are `whitespace`, `constituent`, `single-escape`, `illegal`, `(terminating-macro .` *operation*`)`, and `(nonterminating-macro .` *operation*`)`.

`(skip-whitespace` *`input-stream`*`)` *Operation*

Reads characters from *input-stream* until the next character is not whitespace.

The reader is not sensitive to the case of macro characters.

`(define-macro-char` *`character operation`*`)` *Operation*

Defines *character* to be a reader macro in `standard-read-table`. When *character* is encountered by the reader, *operation* is called with two arguments, the stream and the character that was read.

`(define-nonterminating-macro-char` *`character operation`*`)` *Operation*

Just like `define-macro-char` except that the macro is not triggered if *character* is read inside a token.

There are a number of "quotelike" macro characters present for the convenience of the user.

macro character	*symbol*
'	quote
`	quasiquote
control-v	fluid
control-y	coercer
,@	unquote-splicing
,	unquote

`(define-quotelike-macro-char` *`character object`*`)` *Operation*

Makes *character* a terminating macro which returns a list of *object* and the next thing read. This also arranges for the printer to print using analogous syntax. For instance, the quote syntax is defined with the line `(define-quotelike-macro-char #' 'quote)` in the system internals.

`the-unread-object` *Object*

When a reader macro returns this, the reader takes it to mean that nothing at all was read. For instance, the reader macro for `;` reads the remainder of the line and returns this.

The character `[` is used to read lists in the same way that `(` is, except that `[` must be matched by a `]`. This is mostly for compatiblity with code written at the University of Indiana.

Since there are no packages in Oaklisp, the `:` character is treated like any other constituent.

Most of the Common Lisp hash reader macros are supported. For instance, the character object representing `a` is read `#\a`. Many special characters have long names, such as `#\space`.

`(define-hash-macro-char` *character operation*`)` *Operation*

Defines *character* to be a hash reader macro character. *Operation* should take three arguments: a stream, the character, and the numeric argument that was between the hash and the character, `#f` if none was passed.

There are many hash reader macro characters, including `#o`, `#x`, `#d`, `#b` and `#c` for octal, hexidecimal, decimal, binary and complex numbers, respectively. The syntax `#`*nrxxx* is used to read *xxx* in base *n*. `#(...)` is used for reading vectors. The `#|` macro comments out text until a matching `|#`, with proper nesting. As described in Section 2.7, `#t` and `#f` are read as the canonical true and false values, respectively.

The `#[symbol "..."]` syntax can be used to read arbitrary characters, although the `|...|` construction is prefered. Analogous constructors can be added with the settable operation `hash-bracket-opti`

`(fluid input-base)` *Fluid Variable*

The radix in which numbers will be read.

`(fluid features)` *Fluid Variable*

A list of "features" present in the current implementation, used by the `#+` and `#-` reader macros. Testable and settable with the `feature?` settable operation. It is guaranteed that the `oaklisp` and `scheme` features will be present in any implementation of Oaklisp.

`(fluid current-locale)` *Fluid Variable*

The `#.` macro evaluates its argument in this locale.

`(fluid read-suppress)` *Fluid Variable*

This is true when what is being read will just be ignored, and indicates to the reader that it shouldn't go to the trouble of interpreting the meaning of complex tokens or anything like that.

10.3 Printing

The printer is pretty heavy duty, but has no facilities for printing circular objects.

`(format` *`stream control-string . args`*`)` *Operation*

This is very similar to the Common Lisp `format` function, and is the usual way for users to print things.

Stream is permitted to be `#t` to indicate that output should be sent to the standard output, and `#f` to indicate that the output should be bundled up into a string and returned.

Characters in control-string are printed directly, except for the `~` character which indicates that some action should be taken. The `~` may be followed by a number or by a `:` or `@`, which vary the action that would normally be taken in some way.

Currently defined `~` characters and their associated actions are:

`A` Print and argument with `(fluid print-escape)` bound to `#f`.

`~` Print a `~`.

`%` Do a `newline`.

`&` Do a `freshline`.

`S` Print an argument with `(fluid print-escape)` bount to `#t`.

`B` Print an argument in binary.

`D` Print an argument in decimal.

`O` Print an argument in octal.

`X` Print an argument in hex.

`n``R` Print an argument in base *n*.

`C` Print an argument which is a character.

`P` Print an `s` if the argument is not 1.

`!` Print a weak pointer to the argument, preceded by an expression which evaluates to the argument if `(fluid fancy-references)` is on. This is used to print unique id's for objects without nice printed representations, like operations.

A tilde followed by a newline is ignored; this construct is used for making *control-string* more readable by breaking it across lines.

`(print` *`object stream`*`)` *Operation*

Writes a representation of *object* to *stream*. Users are encouraged to add informative print methods for types they define.

`(define-simple-print-method` *`type string`*`)` *Operation*

Instructs the printer to include *string* in the printed representation of instances of *type*.

`(fluid print-radix)` *Fluid Variable*

The radix in which numbers will be printed. The default is ten.

`(fluid print-level)` *Fluid Variable*

The number of levels of list structure to be printed before the printer abbreviates. The default is `#f`, meaning never abbreviate.

`(fluid print-length)` *Fluid Variable*

The number of elements of a list to be printed before the printer abbreviates. The default is `#f`, meaning never abbreviate.

`(fluid print-escape)` *Fluid Variable*

This controls whether the printer tries to print things that are easy for people to read, or ones that can be read back in to Oaklisp. The default is `#t`, meaning to maintain print/read consistency at the expense of readability.

`(fluid symbol-slashification-style)` *Fluid Variable*

This controls the style of printing of symbols when they are escaped. See the implementation manual for details.

`(fluid fraction-display-style)` *Fluid Variable*

This can be either `normal`, `fancy` or `float`. In these cases, `(/ -5 3)` would print as either `-5/3`, `-1·2/3` or `-1.6666666666`, respectively.

Chapter 11

Miscellaneous

11.1 Tables

The types are `generic-hash-table` and `eq-hash-table`. The access interface is `present?`, which returns a `pair` whose `car` is the key and whose `cdr` is the associated value. A different interface to hash tables is provided by the T-style `table-entry` operation which returns the associated value or `#f` if the key isn't in the table. The setter of either operation can be used to add, modify, and remove associations.

`(make generic-hash-table` *`key-hash-op equal-op`*`)` *Operation*

`(make eq-hash-table )` *Operation*

`(present?` *`table key`*`)` *Settable Operation*
Returns `(`*`key . val`*`)` pair, or `#f` if not present.

`(table-entry` *`table key`*`)` *Settable Operation*
Returns value indexed by *key* or `#f` if not present.

11.2 Delays

Oaklisp's delays are compatible with the facility defined in R3RS, but extend those primitive facilities in two ways. First, the system will automatically force promises when appropriate. For instance, `(+ 2 (delay 3))` does not signal an error; it returns `5`. Similarly, delays are printed transparently, slightly violating read/print consistency. Secondly, the delay facility is user extensible. Users can create new kinds of delays that have special protocols, for instance numeric delays that do not force themselves upon arithmetic operations, but instead make more and more complicated delays.

`(delay` *`expression`*`)` *Special Form*
This immediately returns a *promise* for *expression*, without actually computing *expression*. This promise does not compute *expression* until it is forced, at which point it returns the value of *expression*, computing it if it hasn't already done so.

`(force x)` *Operation*

If *x* is a promise, it is forced to compute its value, which is returned. If *x* is not a promise, it itself is returned.

`promise` *Type*

This is the type of the objects returned by `delay`.

`forcible` *Type*

This is an abstract type, of which `promise` is a concrete subtype. Subtypes of `forcible` are expected to respond to the `force` operation in a sensible fashion. Oaklisp's system internals sometimes force instances of `forcible` automatically, for instance when sending them messages for which no appropriate method can otherwise be found.

`(fluid forcible-print-magic)` *Fluid Variable*

Controls how delays are printed. This is how `(delay 'foo)` would print under various settings of `forcible-print-magic`.

value	print style
`#f`	`#<DELAY 3462>`
`indicate`	`#[DELAY FOO 3462]`
`transparent`	`FOO`

The default is `transparent`. A setting of `indicate` is more instructive if you encounter odd behavior that might be due to delays.

Chapter 12

User Interface

The Oaklisp user interface currently consists of a read-eval-print loop and a simple debugging facility.

Errors land the user into a recursive evaluation loop in which special restart handlers are available. Our implementation includes mechanisms for inspecting objects and tracing function calls.

12.1 The Top Level Environment

All expressions must be evaluated with respect to a particular naming environment. The read-eval-print loop uses the locale specified by the fluid variable `current-locale`. The Oaklisp system boots up with this variable bound to `user-locale`. Other useful name spaces are `scheme-locale`, `system-locale`, and `compiler-locale`.

Several fluid variables are used to keep a short history of the dialogue conducted by the top level evaluator. The most useful of these is `(fluid *)`, which contains the value produced by the most recent user expression. The value of this variable is rolled back into `(fluid **)` and then into `(fluid ***)` to provide access to the three most recent values. Similarly, there are three copies of `(fluid +)` and `(fluid ?)` that provide access to recent expressions that were read in and to their form after macro expansion.

The switch `(fluid fancy-references)` controls the printing of anonymous objects. When this switch is turned off, an object usually prints out something like this: `#<op 806>`. This format indicates the type of the object, and provides a weak pointer that can be derefenced with `object-unhash` to get the object. When the `(fluid fancy-references)` switch is turned on, the printer attempts to generate an expression that will evaluate to the object in the current locale. For example, the above operation might print out as `#<op (setter car) 806>`. The default value for this switch is `#f`, but it is briefly switched on by `describe`.

Two more fluid variables that are frequently used at the top level are `(fluid print-length)` and `(fluid print-level)`, which are normally set to small integer values in order to abbreviate the printing of long lists, but which can be set to `#f` in order to enable exhaustive printing.

12.2 Miscellaneous Functions

There are some other very useful functions that are part of the user interface.

`(apropos` *word [place]*`)` *Operation*

Returns either variables or symbols containing *word*, depending on *place*, which can be a locale or `symbol-table`. *place* defaults to `(fluid current-locale)`.

`(%gc)` *Operation*

Collect garbage. This does not collect garbage in "static space," but it is exceedingly unlikely that there is any there.

`(%full-gc)` *Operation*

Collect more garbage. This does collect garbage from "static space," but more importantly, it put everything not freed into static space, so it need not be transported in future normal garbage collections.

12.3 Debugging

The following special forms can be used to trace the execution of an operation.

`(trace-variable-in` *global-var*`)` *Special Form*

Puts a trace on the operation stored in *global-var*, causing a message to be printed every time the operation is called.

`(trace-variable-out` *global-var*`)` *Special Form*

Puts a trace on the operation stored in *global-var*, causing a message to be printed every time a call to the operation returns.

`(trace-variable-in-out` *global-var*`)` *Special Form*

`(untrace-variable` *global-var*`)` *Special Form*

Objects can be examined in detail with the `describe` function, which prints the object and its type with `(fluid fancy-references)` turned on, followed by the object's internal state. The internal state is organized as instance-variable blocks from the object's various component types. An object's internal state usually contains anonymous objects whose printed representation includes weak pointers which can be dereferenced using `object-unhash`. Together, `describe` and `object-unhash` constitute a simple but effective inspector.

To simplify this process `describe` applied to an integer which is the `object-hash` of some object will describe that object. In other words, `describe` can be applied to the numeric ID in an object's printed representation.

`(describe` *object*`)` *Operation*

Prints out lots of stuff about *object*.

`(object-unhash` *i*`)` *Operation*

Dereferences the weak pointer *i*.

When an error occurs in our implementation of Oaklisp, the user is thrown into a recursive evaluation loop whose dynamic context is nested inside that of the error. Several restart handlers are typically available in a recursive evaluation loop, and the `ret` function is the mechanism for invoking one of these handlers. `call/cc` can be used to preserve an error context when it might be useful to restart the computation at a later time.

`(ret` *n* `.` *args*`)` *Operation*

Invokes restart handler *n*, as specified by the list of handlers printed out by a subordinate evaluation loop. `(ret 0)`, which returns control to the top level evaluation loop, is always in effect.

The following dialogue with Oaklisp illustrates some of these points.

```
Oaklisp 1.0  - (C) 1987 Barak A. Pearlmutter and Kevin J. Lang.
Oaklisp evaluation loop.
  Active handlers:
  0: Return to top level.

> (with-open-file (inf "fone.nums" in) (car (read inf)))

Error: Error opening "fone.nums" for reading.
Oaklisp evaluation loop.
  Active handlers:
  0: Return to top level.
  1: Retry opening file (argument for different file name).
  2: Return to debugger level 1.

>> (call/cc identity)                ;get error context.

>> (set foo (fluid *))               ;stash it away.

>> (ret 0)                           ;back to top level.

   Invoking handler Return to top level..

> (describe foo)                    ;inspect continuation.

#<Op FOO 798> is of type #<Type OPERATION 801>.

 from #<Type 801>:
  LAMBDA? : #<Object 802>           ;what's this thing?
  CACHE-TYPE : 0
  CACHE-METHOD : 0
```

```
  CACHE-TYPE-OFFSET : 0

> (describe (object-unhash 802))

#<Object 802> is of type #<Type %METHOD 803>.

 from #<Type 803>:
  THE-CODE : #<VLmixin 804>
  THE-ENVIRONMENT : #<VLmixin 805>

> (foo 0)                    ;re-enter error context.

>> (ret 1 "phone.nums")      ;resume computation

   Invoking handler Retry opening the file ...

268-7598                     ;got that phone number!

> (exit)

Oaklisp stopped itself...
```

Using the error system effectively is an important part of providing the user with a helpful interface. Details on the error system can be found in Section 6.3.

Bibliography

[1] Software helps city plan trash pickup. *Government Computer News*, 6(18), September 1987.

[2] Henry G. Baker, Jr. Actor systems for real-time computation. Technical Report TR-197, MIT Laboratory for Computer Science, March 1978.

[3] Danny Bobrow et al. Commonloops: Merging common lisp and object-oriented programming. In OOPSLA-86 [9], pages 17–29. Special issue of *ACM SIGPLAN Notices* 21(11).

[4] Adele J. Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[5] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *Transactions of Programming Languages and Systems*, 7(4):501–38, October 1985.

[6] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented Scheme with first class types. In OOPSLA-86 [9], pages 30–7. doi: 10.1145/960112.28701. Special issue of *ACM SIGPLAN Notices* 21(11).

[7] Kevin J. Lang and Barak A. Pearlmutter. Oaklisp: an object-oriented dialect of Scheme. *Lisp and Symbolic Computation*, 1(1):39–51, May 1988.

[8] David A. Moon. Object-oriented programming with flavors. In OOPSLA-86 [9], pages 1–8. Special issue of *ACM SIGPLAN Notices* 21(11).

[9] OOPSLA-86. *ACM Conference on Object-Oriented Systems, Programming, Languages and Applications*, September 1986. Special issue of *ACM SIGPLAN Notices* 21(11).

[10] Barak A. Pearlmutter. Garbage collection with pointers to single cells. *Communications of the ACM*, 39(12):202–6, December 1996. URL http://www.acm.org/cacm/extension/pearlmt.pdf.

[11] Barak A. Pearlmutter and Kevin J. Lang. The implementation of Oaklisp. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 189–215. MIT Press, 1991.

[12] Jonathan A. Rees and Norman I. Adams, IV. T: A dialect of lisp or, lambda: the ultimate software tool. In *ACM Symposium on Lisp and Functional Programming*, August 1982.

[13] Jonathan A. Rees, William Clinger, et al. The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[14] Jonathan A. Rees et al. *The T Manual*. Yale University Computer Science Department, fourth edition, 1984.

[15] Brian Cantwell Smith. Reflection and semantics in lisp. Technical Report CSLI-84-8, Center for the Study of Language and Information, 1984.

[16] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In OOPSLA-86 [9], pages 38–45. Special issue of *ACM SIGPLAN Notices* 21(11).

[17] Guy Lewis Steele, Jr. Lambda: the ultimate declarative. Technical Report AI Memo 379, MIT AI Lab, 1976.

[18] Guy Lewis Steele, Jr. and Gerald J. Sussman. The art of the interpreter. Technical Report AI Memo 453, MIT AI Lab, 1978.

[19] Guy L. Steele Jr. *Common Lisp: The Language*. Digital Press, 1984.

[20] SYMBOLICS. *Symbolics Release 7 Documentation, Volume 2A*. Symbolics, Inc., August 1986.

Index