Oaklisp Summary

Blake McBride (blake@mcbridemail.com)

Startup:

```
    oaklisp  [--world <file>]  [--dump <file>]  [-G]

where:
        --world indicates what world to load. Default is oakworld.bin
                that comes with the system

        --dump provides a file name where a new world will be dumped
               when the system exits

        -G  option to perform a GC prior to dumping a world
```

Exiting the system:

```
    (exit)

    Use ^D to exit error states
```

File types:

File ending	Description
oak	Oaklisp source file
oa	compiled Oaklisp file (fasl)
bin	Oaklisp world — binary representation of an entire Oaklisp system representing all objects in the running environment.

Loading and compiling files:

```
    (load "myfile")  ; will load "myfile.oa" or "myfile.oak"

    (compile-file #*current-locale "myfile") ; compiles myfile.oak to myfile.oa
```

Data Types

Use `(get-type x)` to get the type of an item.

Type	Comment	Example
null-type	null list & false	nil or #f or ,()
truths	true	t or #t
fixnum	decimal	48
&	binary	#b1011
bignum	binary	#2r1011
	octal	#o755
	hex	#xD5
fraction	decimal	2/3
	hex	#xDE/FF
	floats are converted to fraction	3.141
character	letter 'M'	#\M
		#\Space
		#\Newline
		#\Backspace
		#\Tab
		#\Return
		#\Page
symbol	case insensitive (up-cased)	,symb
	case sensitive	,|symb|
	embedded '('	,sy\(mb
string	with embedded "	"hello\" there"
list		,(hello there)
simple-vector		#(3 4 hello)
type	root of all types	type
	root of all objects	object

Saving and loading worlds at runtime:

```
(dump-world "snapshot.bin")  ; save current world, keep running
(load-world "snapshot.bin")  ; replace running world (never returns)
```

The running system can save its entire state with *dump-world* and replace itself with a saved world using *load-world*, which never returns. The files are ordinary world images

usable with *–world*.

List Operations

Type	Comment	Sp	D	Example	Result
car	first element	fast	no	(car ',(A B C))	A
cdr	rest of list	fast	no	(cdr ',(A B C))	(B C)
c????r	shorthand for multiple car & cdr's	fast	no		
cons	add a list node	fast	no	(cons ',A ',(B C))	(A B C)
list	create a list	fast	no	(list ',A ',B ',C)	(A B C)
append	append lists	slow	no	(append ',(A B) ',(D) ',(G H))	(A B D G H)
length		slow	no	(length ',(A B C))	3
nth		slow	no	(nth ',(A B C) 1)	B
first	same as (nth 0 x)				
second	same as (nth 1 x)				
third	same as (nth 2 x)				
etc.					
last		slow	no	(last ',(A B C))	(C)
	Given: (setq lst ',(A B C))				
	replace car cell	fast	yes	(set! (car lst) ',X)	(X B C)
	replace cdr cell	fast	yes	(set! (cdr lst) ',X)	(A . X)

Sp = Speed

D = Destructive

Comments

Semi-colon ';' introduces a comment. Comments extend from the semi-colon to the end of the line.

Evaluation and Quotes

Lists are recursively evaluated. The first element of a list is the function and the remainder are arguments to the function. For example:

```
(func arg1 arg2 ...)
```

Quoting stops evaluation, so

```
(quote (abc def ghi))
```

would not attempt to evaluate *abc*, *def*, or *ghi*. The "'" symbol acts as a shorthand for *quote*, so the following is equivelent to the previous statement.

```
'(abc def ghi)
```

Predicates

function	Returns *NIL*, or *#T* for
null?	null
symbol?	symbols
atom?	non-list items
list?	list items (including *NIL*)
number?	all numbers
integer?	integers
string?	strings
vector?	vectors
eq?	same object
equal?	structurally similar objects or same number

Logical Operators

function	Returns	Example	Comment
not	nil or #T	(not exp)	Logical inverse
and	last exp or nil	(and exp1 exp2 ...)	Stops evalution on first null expression
or	first non-nil exp	(or exp1 exp2 ...)	Stops evalution on first non-null expression

and and *or* can be used as conditionals too. For example:

```
(and exp1 exp2 exp3)
```

is the same as

```
(if exp1
    (if exp2 exp3))
```

Block

(block
exp1
exp2
...)

Execute each expression and return the value of the last expression.

Local Variables

(let ((var1 val1) ; initialize *var1* to *val1*
(var2 nil) ; initialize *var2* to *nil*
...)
exp1
exp2
...)

LET: Creates local variables and executes the expressions in their context. Returns the value of the last expression.

(let* (vars) exp1 exp2 ...)

LET:* Sames as *LET* except the variables are assigned sequentially. Previously defined variables may be used in subsequent variable initializations.

Defining Functions

```
(define (function-name arg1 arg2 arg3 ...)
        exp1
        exp2
        ...)
```

DEFINE: Define function named *function-name* with specified arguments, and run the expressions in that context.

```
(define function-name
        (lambda (arg1 arg2 arg3 ...)
         exp1
         exp2
         ...))
```

Conditionals

In Oaklisp, conditions are considered true if they are any value other than *NIL* or *#F*. Therefore, *NIL* and *#F* are considered as *false*, and all other values are treated as *true* conditions.

```
(if test
    exp1
    [exp2])
```

IF: If *test* expression is *true*, return *exp1*. *exp2* is optional. If *test* is *false*, return *exp2* or *undefined-value*.

```
(cond
    (test1 exp1 exp2 ...)
    (test2 exp1 exp2 ..)
    ...
    (else exp1 exp2 ...))
```

COND: Evalute each *testN* until one is *true*. If a *true* test is found, its expressions are evaluated and the result of the last one is returned. The *else* test is always *true*.

Math Predicates

Function	Description
(number? n)	Is *n* a number
(zero? n)	
(odd? n)	
(even? n)	
(negative? n)	
(positive? n)	
(= a b)	
(!= a b)	
(< a b)	
(> a b)	
(<= a b)	
(>= a b)	

Numeric Operators

Function	Description
+ - * /	Math functions
(abs n)	Absolute value
(1+ n)	
(quotient a b)	integers only
(minus n)	
(floor n)	
(ceiling n)	
(round n)	
(truncate n)	
(modulo a b)	
(remainder a b)	
(expt n p)	
(max a b c ...)	
(min a b c ...)	
(numerator f)	
(denominator f)	

String Predicates

Function	Description
(string? s)	is *s* a string
(= a b)	
(equal? a b)	
(!= a b)	
(< a b)	
(> a b)	
(<= a b)	
(>= a b)	

String Operators

Function	Description
(length s)	
(nth s i)	0 origin
(upcase s)	
(downcase s)	
(reverse s)	
(append s1 s2)	return an appended copy
(copy s)	
(subseq s beg len)	

Macros

```
(define-syntax (add a b)
    „(+ ,a ,b))

(gensym ¸tmp)
```

Oaklisp macros are procedural and unhygienic, more like Common Lisp macros than Scheme's *syntax-rules*. *define-syntax* takes an ordinary expander function that receives a whole form and returns a replacement form; the arguments are destructured like a lambda list and the result is built with backquote/comma. Because expansion is unhygienic, a macro that introduces variables must avoid capture itself, normally with *gensym*. Use *local-syntax* for a macro needed only while compiling the file in which it is defined.

Named let

```
(let loop ((v 0))
    (if (!= v 10)
        (block
            (print v standard-output)
            (newline)
            (loop (1+ v)))))
```

Printing

```
(print x standard-output)
(newline)
(format <stream> <fmt> <arg1> <arg2> ...)
    <stream> = #T = standard-output
               #F = return a string
```

Classes and Types

In classical object-oriented terminology, the following terms apply:

Name	Description
Class	describes the structure and functionality associaed with instances of it (e.g. instance variables and instance methods)
Meta Class	describes the structure and functionality of the class object (e.g. class variables and class methods)
Instance	a unique instance of a class
Object	a class, metaclass, or instance

In Oaklisp, classes and types are the same thing. So, every type in Oaklisp, such as list, symbol, string, number, fraction, etc., are all classes. Also, all types and instances are objects. In Oaklisp, all types and objects are first-class, this means they can be passed to functions, assigned to variables, etc.. They are all treated the same.

Oaklisp has two root classes/types named *Object* and *Type*. *Type* is the root of all types, and *Object* is the root of all objects.

For purposes of this and following examples, the following naming scheme shall be used:

Name	Description
Type	the Type type
type	a type
Object	the Object type
object	an object
iv*	instance variable
cv*	class variable
i*	an instance object
im*	instance method
cm*	class method

Classes – Simple

By “simple” we mean without meta classes.

Creating a type or class:

Format:

```
(define <type-name>
        (make Type <instance variable names> <superclass list>))
```

Example:

```
(define MyClass (make Type '(iv1 iv2) (list Object)))
```

Creating an instance of a type:

Format:

```
(make <your-type>)
```

Example:

```
(define i1 (make MyClass))
```

Inspecting an object:

Format:

```
(describe <object>)
```

Example:

```
(describe MyClass)
(describe i1)
```

Creating instance methods:

Format:

```
(define-instance <method-name> operation)
(add-method (<method-name>
                  (<type name>  <instance variables this method will access>)
                  self  <method argument list>)
            <method code>
            ...)
```

Example:

```
(define-instance imSet-iv1 operation)
(add-method (imSet-iv1 (MyClass iv1) self a)
            (set! iv1 a))

(define-instance imGet-iv1 operation)
(add-method (imGet-iv1 (MyClass iv1) self)
            iv1)

(imSet-iv1 i1 33)
(imGet-iv1 i1)
```

Instance constructor / initialization:

Format:

```
(add-method (initialize  (<type name>  <instance variables being accessed>)
                         self  <argument list>)
                  <initilization code>
                  ...
                  self)
```

Example:

```
(add-method (initialize (MyClass iv1 iv2) self arg1)
     (set! iv1 arg1)
     (set! iv2 44)
     self)

 (define i2 (make MyClass 88))
 ; in this case
 ; i2->iv1 would be 88
 ; i2->iv2 would be 44
```

Classes – Complex

By “complex” we mean with the addition of meta classes. This gives us class variables and class methods.

Creating a type or class:

Creating a type with a meta-type involves the creation of two types; one the meta type and the other the type. The type created must be an instance of the meta type.

Adding initializers and instance methods are as before. The only difference is we now have the ability to define and use class methods and variables. Those will be shown.

Format:

```
(define <meta-type-name>
        (make Type <class variable names> <meta superclass list>))
(define <type-name>
        (make <meta-type-name>  <instance variable names>  <superclass list>))
```

Example:

```
 (define metaMyClass2 (make Type ',(cv1 cv2) (list Type)))
 (define MyClass2 (make metaMyClass2 ',(iv1 iv2) (list Object)))
```

Class methods:

Format:

```
(define-instance <method-name> operation)
(add-method (<method-name>
                  (<meta type name>  <class variables this method will access>)
                  self  <method argument list>)
            <method code>
            ...)
```

Example:

```
(define-instance cmSet-cv1 operation)
(add-method (cmSet-cv1 (metaMyClass2 cv1) self a)
            (set! cv1 a))

(define-instance cmGet-cv1 operation)
(add-method (cmGet-cv1 (metaMyClass2 cv1) self)
            cv1)

(cmSet-cv1 MyClass2 33)
(cmGet-cv1 MyClass2)
(describe MyClass2)  ; to see the contents of the class
```