MIT Concurrent VLSI Architecture Memo 51

Revised Concurrent Smalltalk Manual

Waldemar Horwat

MIT Concurrent VLSI Architectures Group

April 13, 1993

Table of Contents

Chapter 1. Introduction **1**

Implementation 1
Reading Guide 1
Differences from Previous Version 1
Language Changes 1
Compiler Changes 2

Chapter 2. Concurrent Smalltalk **3**

Introduction 3
Functions 3
Extracting Methods 4
Classes 4
Overriding Methods 5
Type Restriction 5
The Class object 5
Local Variables 6
Types 6
Locks 8
Distributed Objects 9
Macros 10

Chapter 3. Concurrent Smalltalk Reference **11**

3.1. Introduction **11**
BNF 12
Methods, Functions, and Primitives 12

3.2. Syntax **13**
Tokens 13
Identifiers 13
Symbols 15
Constants 15

3.3. Programs **17**
Constant Expressions 17
Global Definitions 17

3.4. Classes **19**
Built-in Classes 19
Defining New Classes 19
Class Inheritance 19
Instance Variables 20
Reader and Writer Methods 20
Class Definition Options 20

3.5. Methods and Functions **21**
Introduction 21
Formals 21
Return Values 21
Method and Function Declarations 22
The Calling Process 22
Scoping of Local Variables 22
Functions 23
Methods 23

3.6. Statements **25**
Cfutures 25
Argument Evaluation 25
Application Statement 26
Type Assertion 27
Variable Bindings 27
Multiple Values 28
Syntactic Sugar 28
Flow of Control 29
Loops 30
Primitive Control 31
Returning Values 31
Continuations 31

3.7. Built-in Methods and Functions **33**
Built-in Classes 33
Primitive Selectors 34

3.8. Objects **36**
Objects 36
Copiers 36
Deallocators 36
Class Inquiries 37
Node Numbers 38

3.9. Distributed Objects **40**
Group and Constituents 40
Creation 40
Operations 40

3.10. Logical and Arithmetic Operations **42**
Comparisons 42
Logical Operations 42
Bitwise Logical Operations 43
Arithmetic Operations 43
Floating-Point Operations 45
Character Operations 45

3.11. Lists **46**
Creating 46

3.12. Strings and Arrays **47**
Creating Arrays 47
Operations on Entire Arrays 48
Accessing Arrays 48
Strings 49

3.13. Locks **50**
Lock Operations 50

3.14. Input and Output **51**
Streams 51
Input and Output Streams 51
Creating Streams 51
Reading 52
Writing 52
Formatting 52
Terminal Stream Operations 53

3.15. Macros **54**

3.16. Low-Level Operations **55**
Constants 55

Integers ... 56
Tags ... 56
Physical Segments ... 57
Allocation ... 57
References ... 58
Objects ... 58
Allocation ... 58
References ... 59
Class Inquiries ... 59
Arrays ... 60

3.17. Environment ... 61
Errors ... 61

Chapter 4. Using Optimist II ... 62
Starting the Compiler ... 62
Batch Mode Compilation ... 62
Top-Level Commands ... 63
Utility Commands ... 63
Viewing Objects ... 63
Compiling Programs ... 64
Options ... 65

Bibliography ... 68

Index ... 69

Chapter 1. Introduction

This document describes the Concurrent Smalltalk language as of March 1993. Concurrent Smalltalk evolved significantly since it was originally implemented as part of a Master's thesis in May 1989. Additions and modifications since [3] are marked on the margin; the major differences are summarized later in this chapter. Scott Furman, Todd Dampier, and Shaun Kaneshiro assisted with Concurrent Smalltalk and Cosmos enhancements such as floating-point arithmetic, queue overflow handling, garbage collection, and miscellaneous bug fixes and enhancements. Details on some of these are published in separate documents.

Implementation

To use the features in this document, be sure that you are using the Optimist compiler version 3.0 or later [3] [2], Cosmos version 3.0 or later [5], and MDPSim version 7.2c or later [4].

The Optimist II compiler is written in standard revised Common Lisp as specified in [6]. Unlike previous versions of Optimist II, this one uses the ANSI version of the LOOP macro. Optimist II was developed on a Macintosh using Apple Common Lisp 2.0. It runs on a Macintosh and on Sun Sparcstations using Lucid Common Lisp 4.1.0 or later.

Reading Guide

Chapter 2, **Concurrent Smalltalk**, provides an introduction to the Concurrent Smalltalk language in general. Chapter 3, **Concurrent Smalltalk Reference**, is the specification of the Concurrent Smalltalk language. Chapter 4, **Using Optimist II**, provides a detailed description of the Optimist II features not listed in Chapter 3.

Differences from Previous Version

The following is a list of major changes for the 3.0 version of Concurrent Smalltalk and Optimist:

Language Changes

- Function calls can now be executed on particular nodes. Objects can also be allocated on specified nodes.

- `set` and `cset` on local variables now return `nil` instead of the value being stored to prevent touches when `cset` is used within a `begin`. `set` and `cset` still return the object being modified when used on an instance variable of a specified object.

- Added the `cond`, `when`, `unless`, `clet*`, `let*`, `for`, and `_loop` macros.

- Added the integer `divmod` function.

- Documented floating-point routines.

- Added character and string operations.

- Added support for many low-level operations, including reading and writing object instance variables, low-level object creation primitives, physical segments, MDP global variables, and node number and tag manipulations.

- Array operations and the array class hierarchy are different from those described in earlier documents.

- Documented input/output directives as implemented in the runtime library.

- Futures, `parallel`, and `nparallel` were removed from the language specification; they were never implemented. See the concurrency section in Chapter 2 for a way of guaranteeing concurrent execution.

- Removed the `magnitude` class. Some of the comparisons never worked right and would be inefficient to implement.

- Inline instance variables and classes were removed from the language specification; they were never implemented, and their implementation in Concurrent Smalltalk would be difficult. This is one of the subjects of Aida research [1].

- Removed discussion of declaring user-defined methods for primitive (restricted) selectors such as `*` and `+`, as this is not implemented in Cosmos. See [3] for restrictions that would be in effect if this feature were to be implemented.

- Removed references to the `small-integer` and `large-integer` classes, which were never implemented.

Compiler Changes

- Many compiler register allocation problems and other bugs were fixed.

- Function call messages can now be up to 64 words long.

- Objects can now be arbitrarily long, but only objects small enough to fit in the message queue can migrate.

- Added support for fast context switching used by Cosmos 3.0; Optimist now assumes that register values and the incoming message are destroyed by cfuture faults.

- Code for allocating remote objects is now in the runtime library rather than part of Cosmos.

- Code for looking up methods and allocating closures is now inlined by Optimist rather than being part of Cosmos.

- The code generated by Optimist is compatible with Cosmos changes [5], including three-word headers on all objects, code caching changes in function objects, and new array formats.

- Optimist now generates code for Cosmos's new fast cfuture context switching scheme.

- Object numbering in compiled code is now independent of the machine size. The objects' numbers have their bits scrambled for reasonable distribution on a variety of machine sizes. Also, compiler-generated objects are in a separate namespace from dynamically generated objects.

Chapter 2. Concurrent Smalltalk

Introduction

A Concurrent Smalltalk program is a sequence of top-level definitions. Figure 2-1 shows a sample program that calculates Fibonacci numbers using double recursion.

```
(defmethod fib integer ()
  (if (<= self 2)
    1
    (+ (fib (- self 1)) (fib (- self 2)))))
```

Figure 2-1. A simple Fibonacci program
This program calculates Fibonacci numbers using double recursion. Although it does not use the most efficient algorithm to calculate Fibonacci numbers, it does illustrate Concurrent Smalltalk's implicit concurrency.

The program is a single *method* associated with the *selector* `fib` and *class* `integer`. The fact that the method takes no arguments other than the integer receiver is indicated by the empty list, `()`, on the first line. The following three lines contain the body of the method. `self` represents the *receiver object*, which is the number to which `fib` was applied. The `if` statement checks whether that number is less than or equal to 2. If so, `fib` returns 1. Otherwise, `fib` returns the sum of `(fib (- self 1))` and `(fib (- self 2))`, which are computed concurrently. This concurrent evaluation of arguments is one of the important differences between Concurrent Smalltalk and sequential Smalltalk.

`fib` can be invoked by calling it on an integer (the receiver object):

```
(fib 30)
```

`fib` would then calculate and return the answer `832040`. If `fib` had any more arguments, they would be included after the receiver object, as in:

```
(fib 30 x y z)
```

Functions

The Fibonacci program was defined as a method. It is also possible to define it as a function, as in Figure 2-2. A function is a method not associated with any class or selector. Although in this example methods and functions are equivalent, in other cases, such as in iterators, functions may be more useful than methods.

```
(defun ffib (n)
  (if (<= n 2)
    1
    (+ (ffib (- n 1)) (ffib (- n 2)))))
```

Figure 2-2. A simple Fibonacci program as a function
Functions have no receiver object, so the parameter n has to be specified explicitly.

The syntax for a method and a function call is the same, so `ffib` would also be called by:

```
(ffib 30)
```

The meaning of applying `ffib` to arguments (`30` in this case) depends on whether `ffib` is a selector or a function. If `ffib` were a selector, a method lookup would be done to determine the class of the first argument and then call the method corresponding to the selector and that class, while if `ffib` is a function, it is called directly.

Extracting Methods

A manual method lookup can be done using the `method` primitive. `method` takes two parameters, a selector and a class, and returns a function which performs the same action as the method. For example, the method shown in Figure 2-1 can be extracted using

```
(method fib integer)
```

The result behaves just like the `ffib` function in Figure 2-2. It can be called using

```
((method fib integer) 30)
```

A method extracted in this way does not have to be a direct method of the class; it can be an inherited method.

Classes

A Concurrent Smalltalk class is a type; the two words are used interchangeably in the language definition[1]. A few built-in classes are predefined; these include symbols, booleans, integers, floating point numbers, characters, functions, and other classes. A complete list is given in Table 3-2. All classes are subclasses of the class `object`.

The `defclass` primitive can be used to add user-defined classes. A class definition consists of a list of superclasses and zero or more new instance variables. Each instance object of that class contains those instance variables. The user may also define a number of methods for that class. A simple class that implements Lisp-like lists is shown in Figure 2-3[2].

```
(defclass pair (object) car cdr)

;(defmethod car pair () car)
;(defmethod cdr pair () cdr)
;(defmethod get-car pair () car)
;(defmethod get-cdr pair () cdr)
;(defmethod put-car pair (value):pair (set car value) self)
;(defmethod put-cdr pair (value):pair (set cdr value) self)

(defun cons (first second):pair
  (put-car-cdr (new pair) first second))

(defmethod put-car-cdr pair (first second):pair
  (cset car first)
  (cset cdr second)
  self)
```

Figure 2-3. The `pair` class

The six methods that are commented out by semicolons are defined automatically by `defclass` (in addition to a few others described in Section 3.4). `car` and `get-car` do the same thing; both are defined because `car` is more convenient, but it cannot be used in the body of a method of class `pair` because static scoping shadows the method `car` by the instance variable `car`.

The `:pair` constructs define the result types of the methods. They are unnecessary, but they do improve efficiency and allow rudimentary type checking.

The class `pair` is defined on the first line of Figure 2-3. The `defclass` primitive specifies the class name (`pair`), the superclasses (`(object)`), and the *instance variables* (`car` and `cdr`).

Whenever a class *c* is defined, a *class predicate* and *reader* and *writer* methods are defined automatically, as well other, less-used methods described in Section 3.4. The *class predicate* is a function named *c*? that accepts one argument *a* and returns `true` if *a* is a member of

[1] Nonetheless, the words *type* and *class* have slightly different meanings in the discussion of the compiler in Chapter 3.

[2] This class is similar to the `pair` class predefined in Concurrent Smalltalk's library.

class *c* (or one of its subclasses) and `false` otherwise. Also, for each instance variable *x* of *c*, the methods *x*, `get-`*x*, and `put-`*x* are defined. The first two methods take an instance object *o* as an argument and return the value of *x* in *o*, while `put-`*x* takes two arguments, an instance object *o* and a new value *v* of *x*, and assigns *v* to *x* in *o*. The methods *x* and `get-`*x* are known as *reader methods*, while `put-`*x* is called a *writer method*. The writer methods return *o*, the object to which the value is written.

After a class is defined, additional methods may be defined for it. In the above example, a method `put-car-cdr` is defined for the class `pair`. `put-car-cdr` sets the value of a pair's `car` and `cdr` variables and returns the pair. Inside a method, the receiver's instance variables can be accessed by their names.

Overriding Methods

Consider a class *c2* which is a subclass of *c1*. When a class *c2* defines a method *m2* with the same selector *s* as a method *m1* of *c1*, the class *c2* is said to be *overriding* the method *m1*. When selector *s* is applied to an object of class *c2* or one of its descendants, method *m2* will be used instead of *m1*.

Nevertheless, sometimes it is desirable to call *m1* on an object of class *c2*. For example, method *m2* might want to call the method it is overriding. An overridden method *m1* can be called by performing a manual method lookup using the form `(method` *s* *c1*`)`. The resulting method can be called normally.

Type Restriction

The type of an overriding method must be a subtype of the type of the overridden method. For instance, in the above example the type of *m2* must be a subtype of the type of *m1*. This means that both methods must have the same number of arguments, the types of the arguments of the overriding method must be supertypes (superclasses) of the types of the arguments of the overridden method, and the result type of the overriding method must be a subtype (subclass) of the result type of the overridden method. If any argument of the overridden method is declared inline or using any other declaration, either explicitly or by default, the corresponding argument of the overriding method must have the same type and declarations. The results of violating the above rules are undefined. The compiler may issue errors if the above rule is violated, but it is not guaranteed to do so.

The above restrictions apply only to methods being overridden. There are no restrictions on methods with the same name declared for disjoint classes (i.e. classes which are not subclasses of each other).

The Class `object`

Methods of class `object` are very similar to functions. There are two main differences between functions and methods of class `object`:

- A method of class `object` can be overridden by a method of a more specific class. For example, if `cons` in Figure 2-3 is defined as a function, no other function or method may be called `cons`. On the other hand, if it is defined as a method of class `object`, it may be overridden by a method `cons` defined for integers. However, a method may not be overridden by a function.

- A function that takes no parameters can be defined, while a method must always take at least one parameter—the instance object.

In the interest of code maintenance and readability, it is recommended that functions be used in cases when overriding makes no sense; parameter functions to iterators fall into this category. On the other hand, if overriding a function might be desirable, that function should be defined as a method of type `object`. It is not clear whether overriding `cons` (Figure 2-3) would be useful, so it might be defined either as a function or a method, depending on one's taste.

Local Variables

A method or a function can declare local variables using the `clet` or `let` statements or their derivatives. For example, the function `fib` from Figure 2-1 could be rewritten using two local variables as in Figure 2-4.

```
(defmethod integer lfib ()
  (if (<= self 2)
    1
    (clet
      ((a (lfib (- self 1)))
       (b (lfib (- self 2))))
      (+ a b)))
```

Figure 2-4. Fibonacci program with local variables

The above program is equivalent to the one in Figure 2-1 and actually compiles into the same code.

Local variables declared with a `clet` or a `let` statement have a scope which is the body of the `clet` or `let` statement (except for the bindings themselves). `clet` and `let` statements can be nested. Local variables can be altered using a `cset` or a `set` statement; the difference between the two will be explained in the **Concurrency** section below.

Types

The types (i.e. classes) of various values can be declared explicitly. Such declarations serve three purposes:

- Types allow the compiler to generate faster code by allowing it to perform operations such as method lookup at compile time.
- The compiler can perform type checking to find simple errors such as passing a value of one type to a function that is expecting a value of a different type.
- Declaring types of function parameters and results serves to document the code.

For the purposes of type inclusion, a type is its own supertype and subtype.

Due to the common use of generic types, the compiler's type checking is necessarily limited. In particular, when an expression of type *t1* is assigned to a variable of type *t2* or passed as a parameter to a function that expects type *t2*, the compiler usually will give an error or a warning if *t1* is not *t2*, *t1* is not a superclass of *t2*, and *t2* is not a superclass of *t1*. This does not mean, however, that the semantics of function parameter and return type declarations are any different from their standard interpretations—when a function parameter is declared type *t*, every value passed as that parameter must be a member of type *t*, and when a function result is declared type *t*, the function must return a value that is a member of type *t* as that result—the only difficulty is that the compiler is not able to do full type checking, so it usually follows the rules outlined above.

For example, `integer` and `boolean` are both subclasses of the `object` class (see Figure 3-2), but they are otherwise unrelated to each other. Thus an `integer` can be passed to a function that expects an `object`, an `object` can be passed to a function that expects an `integer`, but a `boolean` cannot be passed to a function that expects an `integer`. The second possibility, passing a more general type to a function that expects a less general one, is included to handle the common case of extracting values from general storage class. One could, for example, keep a `pair` of `integers` and desire to add the pair's `car` and `cdr` together. Since a `pair` is a generic data structure, it can contain values of type `object`; a compiler has no simple way of knowing at compile time that the `pair` will contain `integers`, so the best it can deduce is that the `pair`'s `car` and `cdr` are `objects`.

Types can be declared as follows:

- To specify the type of a local or an instance variable, follow the variable name with a colon and its type. Several locals can be declared using the same type by separating their names with commas.

- To specify the type of a function or method formal, follow the formal name with a colon and its type. Several formals can be declared using the same type by separating their names with commas.

- To specify the result type of a function or method, follow the list of formals with a colon and the result type [1].

- A type of an intermediate result can be specified using a type-assertion statement [2].

The three kinds of declarations are illustrated in Figure 2-5, yet another copy of the Fibonacci program. All untyped variables, parameters, and functions and methods are typed `object` by default.

```
(defun tfib (n:integer):integer
  (if (<= n 2)
    1
    (clet
      ((a:integer (tfib (- self 1)))
       (b:integer (tfib (- self 2))))
      (+ a b)))
```

Figure 2-5. Fibonacci program with types

There are three type declarations here. In order, they are a declaration of the parameter type of `n`, a declaration of `tfib`'s result type, and declarations of the types of the local variables `a` and `b`.

Concurrency

Concurrency is expressed in Concurrent Smalltalk in several ways:

- Concurrent argument evaluation. In

```
(+ (big-computation 3) (time-sink 738))
```

the expressions `big-computation` and `time-sink` can be evaluated concurrently.

- Expressions in `concurrently` and `nconcurrently` statements may be evaluated concur rently.

- The variable bindings in `clet` and `let` statements can also be evaluated concurrently.

For example, the expressions `big-computation` and `time-sink` can be evaluated concur - rently in

```
(cset a (big-computation 3))
(cset b (time-sink 738))
(+ a b)
```

as well as in

```
(let ((a (big-computation 3))
      (b (time-sink 738))
    (+ a b))
```

[1] See also **return values** in Section 3.5 for a description of specifying types of multiple results.

[2] See Section 3.6.

- The computations in assignments using `cset` and in function calls whose result values are unused can be done concurrently with neighboring statements.

Note that, in general, there is no guarantee of concurrent execution—components of `nconcurrently`, `concurrently`, `clet`, and `let` statements may be serialized. There is one way to ensure concurrency when running on a J-Machine: any functions called inside `nconcurrently` and `concurrently` statements or `clet` and `let` bindings whose results are not referenced inside those statements or bindings are guaranteed to run concurrently if these functions are declared `&not-inline`. Note, though, that the interpreter built into Optimist does not support concurrency and will serialize all computations.

The action of a `cset` can be thought of as storing a promise (known as a *cfuture*) to calculate the value of a variable. For example, after

```
(cset a (big-computation 3))
```

is executed, `a` will contain either the value of `(big-computation 3)` or a cfuture promising to deliver that value when it is needed. If `a` contains a cfuture, `(big-computation 3)` is evaluated in parallel by a different task. At the same time, execution of the method can proceed and the method can perform another time-consuming task. It will not have to wait for `(big-computation 3)` to complete until the value of `a` is needed.

Sometimes it is desirable to explicitly wait until the value of an expression is available before continuing. This is called either *touching* the expression. Touching an expression that evaluates to a normal value does nothing. Touching an expression that evaluates to a cfuture causes evaluation to wait until the value of the cfuture is available.

An expression can be touched using the `touch` statement. Since built-in methods and functions usually touch, touching is rarely done explicitly.

The reference manual in Chapter 3 defines more precise semantics for what expressions may or may not be evaluated concurrently.

Locks

```
(defclass resource (object)
  l:lock
  … other fields)

(defmethod init resource ()
  (cset l (new-simple-lock)) ;Creates an initially available lock
  … other initialization code)

(defun new-resource ()
  (init (new resource)))

(defmethod access resource (parameters )
  (acquire l)
  … code to perform the access using parameters …
  (release l))

(defmethod access2 resource (parameters )
  (with-locks (l)
    … code to perform the access using parameters …))
```

Figure 2-6. Lock Example

This example defines a class `resource` that contains a lock. Every call to `access` acquires the lock when it starts and releases it when done, so the code in the middle of the `access` method cannot be interrupted by another `access` method. The `with-locks` macro is a convenient shorthand for acquiring and releasing locks; the `access` method could have been rewritten as `access2`.

Locks are used to synchronize computation by Concurrent Smalltalk programs. Locks are especially useful around critical sections of code where only one process may access a resource; a process that wants the resource acquires a lock before accessing the resource and releases it when it is done. Two variants of locks are provided. `simple-locks` are fast locks which, however, perform poorly when many processes are waiting for a resource; `simple-locks` should be used in situations in which the probability of contention for a resource is small. `queueing-locks` are slower locks designed to handle a large amount of contention.

As an example of the use of locks, suppose one wants to restrict the use of a resource so that only one process can access it at a time. To accomplish this exclusion, a lock can be associated with the resource, in which case every process should acquire the lock before using the resource and release it when done. Figure 2-6 shows sample code used to access the resource.

Distributed Objects

```
(defclass distarray (distobj)
  value)

(defun new-distarray (size:integer)
  (new distarray size)

(defmethod get distarray (index:integer)
  (get-value (co group index)))

(defmethod put distarray (index:integer new-value)
  (set (get-value (co group index)) new-value))

(defmethod size distarray ()
  (logical-limit self))
```

Figure 2-7. Distributed Object Example

This example defines a class `distarray` used for distributed arrays. The `get` method returns the element at position index in the array; since each constituent contains only one element of the array, the `get` method returns the value in the constituent specified by the given index. Similarly, the `put` method routes the message to the constituent specified by index, where it stores new-value. The `size` method simply returns the array's size.

Whereas standard objects serialize messages sent to them[1], distributed objects can accept and process many messages at a time. A distributed object is comprised of an array of constituent objects and a common, group name. When a message is sent to the group name, the operating system routes it to a constituent of its choosing. The constituent can then process the message or send it to another constituent; constituents know how to address each other. The `co` primitive is used to find a particular constituent of a distributed object, while the `group` instance variable can be read to determine the group name of a distributed object given one of its constituents.

For example, a large array might be implemented as a distributed object. When a `get` message is sent to the array to read a value of a particular element, the message is routed to one of the constituents. That constituent examines the given index and forwards the message to the constituent containing the element, which reads and returns the value.

Figure 2-7 shows a simple example of the use of distributed objects to create a distributed array. Each constituent contains only one element of the array to keep this example short; a better implementation would use a `simple-array` at each constituent to reduce the number of constituents needed.

The advantages of using a `distarray` class like the one in Figure 2-7 is that many accesses can be made to the array simultaneously; they do not have to pass through a common bottleneck to access the array. In addition, as will be clarified in Section 3.3, the `get` and `put` methods do not access any instance variables of `distarray` themselves, so they could be in-

[1]Except for a few special cases such as immutable objects and messages which do not need to access an object's data to execute, only one message may be processing on a standard object at a time.

lined wherever they are called[1]; thus, reading or writing the `distarray` in Figure 2-7 could involve only two message sends, which is no less efficient than reading or writing a `simple-array`.

Macros

Concurrent Smalltalk provides a macro facility which can be used to extend the language. A macro consists of a pattern and a replacement. The pattern can contain variables or keywords. If it matches with an expression, that expression is replaced by the replacement, which can be either another pattern or a Common Lisp function[2]. Much of the language itself has been implemented in terms of macros. Figure 2-8 contains a sample macro which defines a `when` form that is the equivalent of a Common Lisp `when`[3].

```
(defmacro (when ?test . ?body)
  (if ?test
    (begin . ?body))
```

Figure 2-8. `when` macro

The `when` form defined by this macro takes a test and a number of statements comprising the body. If the test is true, the statements are executed one after another, as in `begin`. If the test is false, `when` returns `nil`. This macro takes advantage of the fact that `if` returns `nil` if there is no else-clause and the condition is false. The Lisp dot notation is used to indicate that the body forms the rest of the given list.

[1] The compiler's handling of `group` would have to change a little to permit this optimization; the compiler currently treats `group` solely as an instance variable, but there is no intrinsic reason why the compiler could not provide a bypass path that checks whether a method was called on a group ID (as opposed to a constituent ID) and just uses the group ID if it was provided instead of always using the `group` instance variable. When this optimization is implemented, distributed arrays such as the one above will be as efficient as simple arrays.

[2] Concurrent Smalltalk functions may be added as replacements later, when the entire compiler and development system is rewritten in Concurrent Smalltalk.

[3] This macro is defined in the runtime library.

Chapter 3. Concurrent Smalltalk Reference

3.1. Introduction

Concurrent Smalltalk (CST) is a concurrent descendant of Smalltalk. It is an object-oriented programming language developed for multiple instruction/multiple data concurrent computers such as the J-Machine. It is an interesting language for a message-passing concurrent computer because it encourages locality and disciplines the use of message-passing.

Goals

Concurrent Smalltalk is a high-level language intended for general-purpose programming of the J-Machine. It was created and revised with the following goals in mind:

- **Expressiveness.** Concurrent Smalltalk must be expressive enough to support the parallel programming paradigms we desire to research on the J-Machine. In particular, it must support object-oriented programming and fine-grained parallelism. Also, since a large part of the Concurrent Smalltalk runtime system is written in itself, Concurrent Smalltalk must support higher-order features such as reasoning about classes of objects.

- **Consistency.** Features which would interact destructively with other features were left out. For example, `become`, although a useful Smalltalk-80 construct, would confuse the type semantics so it was left out.

- **Simplicity.** Concurrent Smalltalk should be as simple as possible. In order to reach the goal of simplicity, Concurrent Smalltalk should consist of a few orthogonal concepts. It is very important that Concurrent Smalltalk contain *no surprises* —one should be able to tell what a program should do by reading it. Features involving action at a distance (i.e. having a statement invisibly affect another statement far away) were intentionally excluded.

- **Familiarity.** Programmers familiar with existing languages should be able to carry over their experience to Concurrent Smalltalk. Also, corresponding features should act in the same ways, which reinforces the "no surprises" philosophy. On the other hand, Concurrent Smalltalk is most similar to Smalltalk-80, Common Lisp, and Scheme in this respect. Hence, static scoping is used for variables.

- **Efficiency.** It is important to be able to compile Concurrent Smalltalk programs into efficient machine code. An efficient implementation allows a programmer to concern himself primarily with algorithms and implementation rather than performance tuning. Concurrent Smalltalk is not a tightly bound low-level language in order to give the compiler latitude in optimizing code.

- **Commonality.** The sets of built-in classes and methods presented in this language specification are by no means minimal. However, the built-in classes are frequently used and were included in order to provide a common base for Concurrent Smalltalk programs. The inclusion of frequently used classes has three advantages:
  - The built-ins are implemented only once, saving time and effort.
  - The built-ins provide a consistent functional and naming specification.
  - The built-ins can be optimized for efficiency.

Format

BNF

The syntax of commands is presented in BNF. Literals are presented in bold, while non-terminals and metasymbols are plain. There are two enhancements to the BNF syntax:

The { *expr1* | *expr2* | … | *exprn* } form specifies that each *expr* can appear at most once, but they can appear in any order. The *symbol* ≡ *expr* form is a macro used for readability. It specifies that whenever *symbol* appears, it should be replaced by *expr* before any productions are done.

Methods, Functions, and Primitives

The declarations of methods and functions are presented in a syntax similar to that used by `defmethod`. To give an example,

(move *what*:robot *x*, *y*, *z*:integer *theta*:float **)** :result **Method**

declares a method called `move` of class `robot` that takes a receiver argument *what* of class `robot`, three `integer` arguments, *x*, *y*, and *z*, and a `float` ar gument, *theta* . That method returns an object of class `result`.

Sometimes an *abstract class* like `number` is declared that has no direct instance objects; instead, every object of class `number` is also an object of one of `number`'s subclasses. Methods of an *abstract class* may or may not have definitions for that class. A method that does not have a definition for the ab stract class is called an *abstract method* . For example, + is an abstract method of class `number`; there exists no generic method to add two arbitrary numbers. Instead, when + is called on two numbers, the definition of + for either the class `integer` or the class `float` is used. Were a third `number` subclass, `complex-number`, defined, it would have to define its own + method. On the other hand, the `zero?` method of class `number` is not abstract because it uses the = method (a method defined on all `numbers`). Thus, `complex-number` does not have to define its own `zero?` method.

The right side of the declaration line indicates the kind of entity being described. Abstract methods are indicated by the words **Abstract Method** . Primitives and macros are indicated by the word **Primitive** or **Macro** ; they are not first-class values and cannot be manipulated from within the language. To use a primitive such as `eq` as a first-class function, declare a new function `feq` that calls `eq` on its two arguments.

3.2. Syntax

Tokens

A Concurrent Smalltalk token is an arbitrarily long string composed of the characters `A-Z`, `a-z`, `0-9`, `_`, `!`, `?`, `%`, `+`, `-`, `*`, `/`, `.`, `<`, `=`, `>`, `&`, `@`, and `^`. The characters `!`, `?`, `&`, and `@` may not be used at the beginning of a token, and a token may not be composed entirely of periods (`.`) or underscores (`_`). Also, tokens beginning with an underscore (`_`) or a percent sign (`%`) are reserved for system purposes and macros and should not be used by user programs. Case is not significant.

A token is considered to be a number if it consists entirely of the characters `0-9`, `_`, `+`, `-`, `/`, `.`, `E`, or `I`; it contains at least one digit; it begins with `+`, `-`, or a digit; and it does not end with a digit. These rules are borrowed from Common Lisp. `E` introduces an exponent, while `I` can be used for complex numbers if they are implemented. Any token that is not a number is an identifier.

Identifiers

Concurrent Smalltalk uses static scoping of identifiers. Local identifiers shadow identical global identifiers, and the meaning of an identifier can be determined by its location in the text of the program. Global identifiers are introduced by the following top-level statements and their derivatives:

- `defconstant`, to define a constant;
- `defglobal`, to define a global variable;
- `defclass`, to define a class;
- `defselector`, to define a method selector;
- `define`, to define one global identifier in terms of another.

The syntactic sugar `defmethod` expands into, among other statements, a `defselector`, defining a global identifier. Similarly, `defun` expands into a `defconstant` statement that defines the function.

Except for classes, the above categories share a single name space. Redefining a global identifier causes an error or a warning unless the new definition is identical to the old one. Class names have lower precedence than other global identifiers, so a global constant can shadow a class name.

All macros are global; however, macros are also in a name space separate from the one shared by the above categories. Since macros match patterns instead of just names, two macros may share the same name. If more than one pattern is applicable, one is chosen at the implementation's discretion. Whenever a macro is applicable, it is expanded, unless one of the literals specified in the macro pattern is shadowed by a local identifier.

Local identifiers are introduced by the following statements and their derivatives:

- `lambda` and `defun` introduce the names of formal parameters.
- `method-lambda` and `defmethod` introduce `self`, `group`, names of the instance variables, and names of the formals. If a name conflict occurs between the formals, instance variables, `self`, and `group`, the results are unspecified.
- `let`, `clet`, `mv-clet`, and `mv-let` introduce names of the locals.
- `lambda`, `method-lambda`, `defun`, `defmethod`, `block`, and `loop` introduce the names of continuations.

All of the shadowing rules are summarized in Figure 3-1.

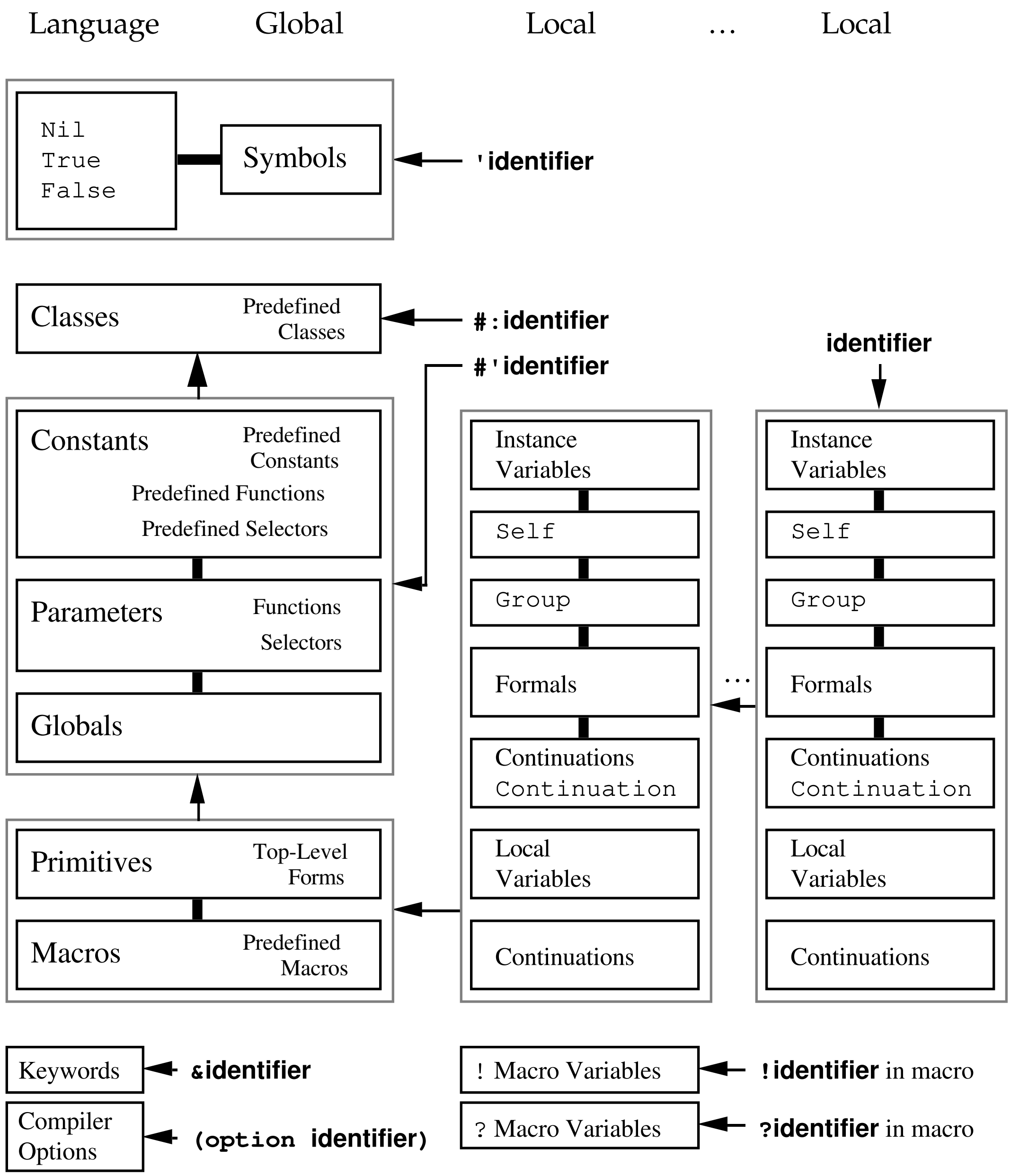

Figure 3-1. Scopes of Identifiers

The scopes of various kinds of identifiers are shown above. Except for macros, sets of identifiers connected by thick lines are mutually exclusive and may not contain duplicate names. To find the meaning attributed to an identifier, follow the arrows from the bold pattern indicating the identifier's usage to the first box that contains the identifier. For example, if `i` is encountered in a program, it is first checked to be a local in the innermost scope, then a local in the next innermost scope, and so on until the global scope is reached. If `i` is not a valid macro pattern, it is checked against the globals, parameters, and constants, and finally classes. On the other hand, if `#:i` is encountered, `i` is checked against the names of classes only. `#'i` searches only globals, parameters, and constants, both user and predefined.

Identifiers that are not defined globally[1] or in any enclosing scope are defined as globals. They must be defined before they are used. The exceptions to this rule are identifiers enclosed in `quote` or `class` statements listed below.

(global *identifier***)** **Primitive**
#'*identifier*

`global` returns the global identifier *identifier*, which, if already defined, must be a global (not a class). If *identifier* is not already defined globally, it is defined as a global.

(class *identifier***)** **Primitive**
#:*identifier*

`class` returns the global class *class*. Since classes are in a separate name space from other globals, no error occurs if there is already a global identifier defined with the same name as *identifier*.

Symbols

(quote *constant* **)** **Primitive**
'*constant*
constant ::= **nil** | **true** | **false** | *identifier* | *number* | *character* | *string* | *list*
list ::= **(***constant* ***)**

Symbols can be specified by preceding including them in a `quote` form as above, which can be abbreviated by a quote mark (`'`). When presented with an identifier, the `quote` expression evaluates to a symbol. Any valid identifier except `nil`, `true`, and `false` can be used—symbols cannot be captured by any scope, nor can they be globally redefined. `nil`, `true`, and `false` are treated specially— `(quote nil)` returns the null object `nil`, while `(quote true)` returns the boolean `true` and `(quote false)` returns the boolean `false`. `(quote` *number*`)`, `(quote` *character*`)`, and `(quote` *string*`)` just return the number *number*, *character*, or *string*.

`quote` also accepts lists of constants (which may be other lists) in parentheses. Any such lists are constructed using `cons`. An empty list generates `nil`.

Constants

A few constants are predefined. Some of these are listed in Table 3-1 below. In addition, any number can be specified by just including the number. Characters can be specified by preceding them with `#\`. Strings can be specified by enclosing them in double quotes (`"`). Double quotes can be included inside strings by preceding them with `\`.

Table 3-1. Predefined Constants

Constant	Value	Class
`TRUE`	`true`	`true`[2]
`FALSE`	`false`	`false`[3]
`NIL`	`nil`	`null`

---

[1] It is up to the implementation to define the meaning of a global definition here. When a file is compiled, an implementation might choose to read all of the definitions in the file and then compile the code, or it could compile the file incrementally. In the latter case forward-referenced identifiers will be considered undefined.

[2] Since `true` is a global constant, `#:true` has to be used to refer to class `true`. Also, class `true` is a subclass of class `boolean`.

[3] Since `false` is a global constant, `#:false` has to be used to refer to class `false`. Also, class `false` is a subclass of class `boolean`.

Comments

Comments may be placed anywhere in source files. A comment starts with a semicolon (`;`) and is terminated by the end of the line. Comments are treated as if they were line breaks by the reader.

3.3. Programs

program ::= (*top-level* | *statement*)*

A Concurrent Smalltalk program is a sequence of top-level forms. Additionally, an implementation may allow `begin` and `if` as top-level forms if *test* is a constant expression and statements in *body*, *consequent*, and *alternative* are top-level forms. Other statements may also be allowed at the top level by extended implementations. Statements at the top level are executed sequentially as if they were enclosed in a `begin`.

Constant Expressions

Constant expressions are expressions that have to be evaluated at compile time. A constant expression can include any expression or function call, except that constant expressions may not produce distributed objects as values and may not call functions that use futures.

Global Definitions

All constants, parameters, and globals reside in a single name space; in general, redefining an identifier with a different meaning causes an error. Macros reside in a separate name space and do not conflict with each other or any other global objects (although they may be shadowed by local static scoping).

(defconstant *name*[**:***type*] *value***)** **Top-Level Primitive**

`defconstant` defines a constant named *name*. The constant can be any valid Concurrent Smalltalk type. If *type* is specified, the value must have that type. Once a constant is defined, it may not be changed (another constant is accepted, though, if it has the same value). Constants encountered in methods are replaced by their values at compile time. *Value* must be a constant expression. Predefined constants are listed in Table 3-1.

Language primitives and built-in functions and selectors are defined as global constants.

(defparameter *name*[**:***type*] *value***)** **Top-Level Primitive**

`defparameter` defines a parameter named *name*. The parameter can have any valid Concurrent Smalltalk type. If *type* is specified, the value, if present, must have that type. If no *type* is specified, *type* is assumed to be `object`, the most general type. Parameters encountered in methods are replaced by their values at compile time. *Value* must be a constant expression. Unlike constants, parameters may be redefined at the top level, but their types may not be changed. The value of a parameter may not be changed by a running program.

User functions and selectors defined using `defun`, `defmethod`, and `defselector` are defined as parameters. Hence, they may be redefined.

(defglobal *name*[**:***type*] [*value*]**)** **Top-Level Primitive**

`defglobal` defines a global named *name*. The global can have any valid Concurrent Smalltalk type. If *type* is specified, the value, if present, must have that type. If no *type* is specified, *type* is assumed to be `object`, the most general type. *Value* must be a constant expression.

A global may be defined several times, but only the *value* from the first definition is used. Nevertheless, all definitions of a global must have the same *type*.

(define *name name***)** **Top-Level Primitive**

This primitive defines the first name as an alias for the object specified by the second name. For example, if the second name refers to a global, after this primitive is executed, both names will refer to the same global.

(undef *name***)** **Top-Level Primitive**

This primitive removes the top-level definition of name, if any. It should be used with caution, as it is possible to bring the system into an inconsistent state using `undef`.

3.4. Classes

Built-in Classes

A few classes are predefined. These are listed in Table 3-2, and their hierarchy is shown in Figure 3-2. The `defclass` primitive can be used to define other classes, which may be based on the built-in ones shown in bold in Figure 3-2.

Defining New Classes

(defclass *class* { *class-declaration* } *superclasses* *instance-var-spec****)** **Top-Level Primitive**

class ::= *name*
superclasses ::= **(***class*+**)**
instance-var-spec ::= *typed-names* | **(***typed-names* { *instance-var-declaration*}**)**
typed-names ::= *name* (**,** *name*)* [**:** *type*]
instance-var-declaration ≡ **&reader** *names* |
&writer *names* | **&cwriter** *names* |
&cas-er *names*
names ::= *name* | **(***name****)**

`defclass` defines a new Concurrent Smalltalk class. A class is a template for specifying objects and methods. Each object belonging to the class contains the instance variables defined in the class definition as well as the instance variables inherited from its superclasses, if any.

In the class definition, *class* is the new class name. It is followed by an optional declaration, described later, the class's superclass list, and finally the additional instance variables declared by the class.

Class Inheritance

Each user-defined class must have at least one superclass, but it may have more than one. A class inherits the instance variable and method definitions from its superclasses. It may add its own instance variables and methods, and it may attempt to override existing methods. If a class is overriding a method, the new method must be a subtype of the existing one.

A simple form of multiple inheritance is allowed. Two or more superclasses may be specified for a class under the following conditions:

- There must be no instance variable conflicts among the superclasses. Formally, this requirement is satisfied if and only if out of the superclasses $s_1, s_2, \ldots s_n$ provided there is one s_i such that if v is an instance variable of s_j, $1 \le j \le n$, then v is an instance variable of s_i or one of its superclasses.

- There must be no inherited method conflicts among the superclasses. Formally, this means that if selector s is associated with method m_i for superclass s_i and method m_j for s_j, then m_i and m_j are the same method (Textual equivalence of the method code is not enough; m_i and m_j must "point" to the same method).

The class then inherits all of the instance variables and all methods from all of its superclasses.

Instance Variables

After the superclasses in the class declaration is a list of new or redefined instance variables. Instance variables without any type are given the type `object`. An instance variable may be specified to have the same name as an instance variable of one of the superclasses. If so, the specified type must be a subtype of the original instance variable's type, and either both or neither must be inline.

Reader and Writer Methods

A few methods are automatically defined when a class *c* is defined. For each instance variable *x* of *c*, two functionally identical methods are defined, named *x* and `get-`*x*, that, when called on an object *o* of class *c*, return the value of *x* in *o*. These methods are called *reader* methods; two are defined in order to avoid name conflicts with instance variables. Similarly, a *writer* method `put-`*x* is defined that, when called on an instance object *o* of *c* and a new value *v* of *x*, and assigns *v* to *x* in *o* and returns *o*. Furthermore, a *cwriter* method `cput-`*x* is defined that behaves just like the writer method `put-`*x* except that it is not strict—it does not necessarily touch its second argument *v*. Finally, a *cas-er* method `cap-`*x* is defined that performs an atomic compare-and-put operation: `(cap-`*x* *o* *comparison* *replacement*`)` checks whether the value of instance variable *x* in *o* is `eq` to the value of *comparison*. If so, it stores the value of *replacement* in *x* and returns `true`; otherwise, it returns `false`.

If it is desirable to produce reader, writer, cwriter, or cas-er methods with names different from the defaults, the `&reader`, `&writer`, `&cwriter`, and `&cas-er` options can be used to specify the new names. More than one method name may be specified for an instance variable. If `&reader`, `&writer`, `&cwriter`, or `&cas-er` is used, the corresponding default method name is not defined. For example, if `&writer` is used with an empty list of names, the corresponding writer method name is not defined.

Class Definition Options

class-declaration ≡ **`&immutable`** |
`&predicate` *names*

A class definition allows a couple of options which are described in more detail below.

Objects of an *immutable* class declared with the `&immutable` option may be shallow-copied at any time at the system's discretion, which can lead to significant performance improvements. They are also often passed by value to methods and functions. It is not necessary that no methods ever write to instance variables, but only that the effects of such writes not be visible outside the class data abstraction. The compiler is free to ignore `&immutable` declarations.

The `&predicate` option defines the name or names of the class predicate. A class predicate is a function that returns `true` when called on an object of the specified class or its subclasses and `false` on all other objects. The default name of a class predicate is obtained by concatenating a question mark (`?`) to the end of the class name, so `(integer? x)` tests whether `x` is an `integer`.

3.5. Methods and Functions

Introduction

Methods and functions are the basic blocks of computation in Concurrent Smalltalk. Each method and function can accept a number of arguments, which are assigned to the formals for the duration of the execution of the body of the method or function. Furthermore, a method has a special first argument, called the receiver, which contains an object of the method's class on which the method was called. In general, methods and functions execute concurrently unless explicitly synchronized. This is true even if they are accessing shared objects.

Formals

formal-spec ::= *typed-opt-names* | **(***typed-opt-names* {*formal-declaration*}**)**
typed-opt-names ::= *opt-name* (**,** *opt-name*)* [**:** *type*]
opt-name ::= *name* | **_**
formal-declaration ≡ **&value** | **&no-leak** | **&name** *name*

A method's or function's formals are listed when the method or function is declared. Each *name* specifies the name of a formal. *Typed-opt-names* specifies one or more names separated by commas followed by an optional type. The character _ can be used to indicate an unnamed formal; unnamed formals accept arguments but cannot be referenced from within the method or function. If *type* is not present, it defaults to `object`. If the long form of a *formal-spec* is used, the formals in *typed-opt-names* can be declared using declarations.

Arguments are passed by value, just as in Smalltalk-80, Scheme, and Common Lisp. The types of the arguments to the method or function must be subtypes of the types of the corresponding formals. A method or a function may assign a value to a formal, which only changes the method's or function's local value. Of course, a method or a function is also free to mutate a formal using some other method; such changes *are* visible to the outside. This kind of mutation corresponds to communication via shared objects.

A formal may be declared `&value`, which means that, at the implementation's discretion, the method or function may be passed a shallow copy of the argument when it is called. Thus, not only is the formal passed by value, but its first-level structure may also be passed by value. All formals declared using an `&immutable` class are automatically declared `&value`.

Declaring a formal `&no-leak` is a hint to the compiler that the value of the formal is not passed out of the method or function, and it will not be referenced after the method or function returns. Thus, the implementation is free to perform a shallow deallocate on the value of the formal when the function returns. This declaration is especially helpful for arguments of type `funct`. The compiler is free to ignore this declaration.

`&name` can be used to name an anonymous function or method. The name is saved for debugging purposes. `&name` is only allowed in a `lambda` or a `method-lambda`.

Return Values

return-specs ::= **:** *type* | **::** **(***return-spec*****)**
return-spec ::= *typed-names* | **(***typed-names* {*return-declaration*} **)**
typed-names ::= *name* (**,** *name*)* [**:** *type*]
return-declaration ≡ **&value**

A method's or function's return specification may be listed when the method or function is declared. Most methods and functions return only one value. For these functions, the short form, consisting of a colon (`:`) followed by the return type, is adequate. If the return type is `object`, the entire return specification can be omitted altogether.

The long form of declaring a method's or function's return types uses the double colon (`::`) notation and allows explicit naming of the return continuation. The name is called a *continuation name*. Continuation names are lexically scoped and may be referenced in the body of the method or function. The syntax and semantics of continuation declarations are analogous to those of formals, and the continuation names reside in the same namespace as formal and variable names. The only declaration allowed is `&value`. If the short form is used, a default continuation name `continuation` is used. Some implementations may also allow returning multiple values. Multiple values do not all have to be returned at the same time, but all have to be returned at most once by the time the method or function finishes.

Since the implicit `return` statement at the end of a method's or function's body returns its value to `continuation`, it is an error to allow execution to "fall through" the method or function to the implicit `return` statement unless one of the continuations is named `continuation`.

Method and Function Declarations

funct-declaration ≡ **`&non-strict`** |
(**`&inline`** | **`&not-inline`**) |
`&side-effect-free`

The following declarations are allowed for methods and functions:

- The `&non-strict` declaration specifies that the arguments do not have to be touched before the body of the method or function begins executing. Thus, the method or function may at the compiler's discretion receive cfutures in the formals. This declaration is useful mainly for inline functions.

- The `&inline` and `&not-inline` declarations specify that the method or function should or should not be included inline at the points where it is called. This declaration is only a hint, and the compiler does not have to obey it.

- The `&side-effect-free` declaration is a hint to the compiler that the method or function does not perform any visible side effects on its arguments or on the global environment. This information lets the compiler better schedule calls to the method or function. This directive is also useful on methods and functions that <u>do</u> perform side effects; it tells the compiler that those side effects are not essential. One example of a method that falls into this category is a method operating on an immutable class of complex numbers that allows redundant representations in rectangular or polar form. The method could side effect a complex number to calculate its polar representation from its rectangular one, but that side effect is not essential for the program to work correctly.

The Calling Process

When a function or a method is called, the values of the arguments are computed and assigned to the formals. The formals are touched unless the function is declared `&non-strict`. After all formal values are evaluated, execution of the method's expressions proceeds as if the expressions were enclosed in an implicit `block`—initially the first expression is evaluated, then the second one, and so forth. The value of the implicit `block`, which is the value of the last expression, is returned to the caller unless an `exit` or `return` statement is encountered first.

Scoping of Local Variables

Local variables are statically scoped. Any `lambda` or `method-lambda` created within a method or a function is a full closure and may reference and alter the method's or function's local variables. Similarly, the method or function may alter its locals, and such changes will be visible to any `lambda` or `method-lambda` nested within it.

If concurrency and efficiency are desired, however, such sharing should be avoided whenever possible. A `lambda` or `method-lambda` should declare its own temporaries for local computations instead of using ones belonging to an outer static scope. If a method or function wants to pass values into a closure, it should initialize the appropriate temporaries before the closure is created and not change those temporaries afterwards. The closure should not change those temporaries either, unless it wants to pass a result back to the method or function that created it.

Functions

(lambda **(***formal-spec*****)** [*return-specs*] {*funct-declaration*} *body* **)** **Primitive**

`lambda` defines and returns an anonymous function. *Formal-spec** is a list of the function's formals and their types. *Return-specs* specifies the function's return type, or, if it returns multiple values, the number of return values and their types. The function may also have declarations, as explained above. *Body* is a list of statements that form the body of the function.

(defun *name* **(***formal-spec*****)** [*return-specs*] {*funct-declaration*} *body* **)** **Top-Level Macro**

`defun` defines a global function with name *name*, formals as specified in *formal-spec**, return values defined by *return-specs*, optional declarations *funct-declaration*, and body *body*.

Methods

(method-lambda *class* **(***formal-spec*****)** [*return-specs*] {*funct-declaration*} *body* **)** **Macro**

`method-lambda` returns a method of class *class*. The resulting method does not have a selector. Nevertheless, it can be called as a function if the first argument is an instance object of *class*. The other parameters are as in `lambda`.

Method-lambda also introduces into the scope of *body* the names of the instance variables of an object of class *class* as well as two special variables: `self` and `group`. `self` refers to the first argument of the method call, also known as the receiver object. If *class* is a subclass of `distobj`, `group` refers to the group name of the distributed object of which `self` is a constituent.

(defselector *selector***)** **Top-Level Primitive**
selector ::= *name*

`defselector` defines *name* as a selector. This primitive is rarely used explicitly, as all undefined names are assumed to be selectors by default.

(add-method *selector class value* **)** **Top-Level Primitive**

`add-method` associates a method with its class and selector. When *selector* is called with a receiver object that belongs to *class*, *value* is called. *Value* should be a function or a method.

(method *selector class***)** **Primitive**

`method` performs the inverse of the `add-method` operation—it returns the method associated with *selector* and *class*. If no method is associated with *selector* and *class*, `method` returns `nil`.

(defmethod *selector class* **(***formal-spec* ***)** [*return-specs*] **Top−Level Macro**
{*funct-declaration*}
body **)**

`defmethod` defines a global method with the given selector and class. The rest of the syntax is anal ogous to `defun`.

When a method is called, the values of the selector and arguments are computed, and the method associated with the selector and the class of the re ceiver object is found. Of course, this method may have been defined for a superclass of the class of the receiver object (i.e. it may be inherited). It is an error occurs if no such method exists. Otherwise, the process of calling a method is the same as that of calling a function.

3.6. Statements

value ::= *statement*
expression ::= *statement*
body ::= *statement**

In the definitions below, the non-terminals *value*, *expression*, and *statement* all refer to statements, although *value* usually denotes a side-effect-free statement that is executed for its return value, *expression* denotes a statement that may have side effects but is executed mainly for its return value, and *statement* denotes a statement that is executed mainly for its side effects. A *body* is a sequence of statements executed one after another just like in `begin`; the value of a body is the value of its last statement.

Cfutures

Cfutures (context futures) are the main means of achieving concurrency in Concurrent Smalltalk. A cfuture is a promise to produce some value at a later time. Cfutures are generated by almost all primitive operations. *touching* a cfuture forces it to calculate and return its value. Cfutures are always *eager*—if left alone, they will tend to evaluate to their values. Cfutures are intended to be very cheap—they can be created and touched in a few assembly language instructions.

Note that, unlike MultiLisp futures, cfutures are not guaranteed not to be touched—they may be touched at any time at the compiler's and operating system's discretion. In an extreme case, cfutures may be touched as soon as they are created, leading to a sequential implementation of Concurrent Smalltalk[1].

Cfutures may have values of complicated expressions as their promises. For example, if `(f 3)=30`, `(g 7)=49`, and `(h 30 49)=79`, during the execution of the statement

```
(cset a (h (f 3) (g 7)))
```

`a` may be computed in arbitrary order, and `f` and `g` need not have returned values by the time the next statement is executed. If `a` is later touched, it will assume the value `79`.

The semantics become more complicated if the functions `f`, `g`, and `h` have side effects. The order of evaluation of arguments of function calls is undefined and may be parallel, so `f` and `g` may be evaluated in parallel. Furthermore, if `h` is declared `&non-strict` (as many built-ins are), the evaluation of `h` may overlap with the evaluation of its arguments. If, say, `h` does not use the value of its second argument until late in its execution, `h` may already be executing while `(g 7)` is still being calculated. Finally, if `h` can return without ever requesting the value of its second argument, `(g 7)` may never be completely evaluated (since cfutures are eager, it will keep evaluating, but the entire program may finish before it is done). A good example of this phenomenon is `(and b false)`, where the program can proceed without ever determining the value of `b`.

Argument Evaluation

Unless a method or a function is declared &non-strict, method and function calls are strict with respect to cfutures—the arguments of a method or function are guaranteed not to be cfutures when the method or function begins evaluation. For example, in

[1] However, the J-Machine implementation guarantees concurrency in some cases, as outlined in the concurrency section in Chapter 2.

```
(cset a (h (f 3) (g 7)))
(cset b (k 10))
(touch a)
(cset c (l 10))
```

(f 3) and (g 7) are guaranteed to be done evaluating before (h 30 49) begins evaluating. Also, (f 3), (g 7), and (h 30 49) are guaranteed to be done evaluating before the evaluation of (l 10) is started. However, (k 10) can be evaluated concurrently with any of (f 3), (g 7), (h 30 49), or (l 10).

The arguments of functions are evaluated concurrently. This means they may be evaluated sequentially, in parallel, or any combination of the two. Using side effects can sometimes lead to deadlock. For example, suppose that the function `release-lock` releases a global lock and `acquire-lock` waits until the lock is released and then acquires it. Further, suppose that `global-lock` is originally acquired. Then, the expression

```
(h (release-lock global-lock) (acquire-lock global-lock))
```

can lead to deadlock because the implementation might choose to evaluate `acquire-lock` sequentially before `release-lock`.

Concurrent evaluation order is also distinct from an arbitrary sequential order. For example, suppose that `c` is a local variable with an initial value of 0 and consider the value of the expression

```
(cset a (+ (begin (cset c (+ 1 c)) c) (begin (cset c (+ 1 c)) c)))
(touch a)
(touch c)
```

Under sequential evaluation of arguments, the final value of `a` would always be 3 and the final value of `c` would always be 2 when this expression completes. Under concurrent evaluation of arguments, the final value of `c` could be 1 if, say, both increments were done before either assignment to `c`. In this case, `a` would get the value 2.

(touch *expression***)** **Primitive**
(touch *expression****)** **Macro**

If *expression* is not a cfuture, `touch` does nothing. Otherwise, `touch` waits until the value of the cfuture is available and then returns that value. It should be kept in mind that if `touch` is used in a subexpression, other subexpressions may or may not continue evaluating while this `touch` is waiting. Also, a `touch` in a subexpression does not guarantee that the entire expression will not yield a cfuture, as is demonstrated in one of the examples above.

If more than one expression is specified, `touch` touches them all and returns the value of the last one. If no expressions are specified, `touch` returns `nil`.

Application Statement

(*funct arg****)** **Primitive**
(*funct* **@***node arg ****)** **Primitive**
funct ::= *expression*
node ::= *expression*
arg ::= *expression*

The first item of an application statement is either a method selector or a function. If it is a selector, the method corresponding to the selector and the class of the first argument is called using the arguments provided. If it is a function, it is applied to the specified arguments. The first item can also be any expression that evaluates to an object of type `funct`. The value of the application statement is either the return value or a cfuture promising that value.

The order of evaluation of arguments is not specified; in fact, some of them may be (but are not guaranteed to be) evaluated concurrently. The arguments are not guaranteed to be touched before being passed to the funct—some of them may be passed to the funct as futures or even cfutures (However, all user-defined methods and functions not explicitly declared `:non-strict` will touch their arguments before their code begins executing). For example, `(cset a (+ 0 a))` does not touch `a`, and `(and b false)` does not touch `b`.

The node number where the function or method is to be executed can be specified explicitly as *node*. The J-Machine implementation will obey this directive and run the function or method on the node with NNR equal to *node*, which must be an integer and a valid node number (see Section 3.8). If no *node* is specified, the function or method is generally called on the node containing the first argument or a random node if there are no arguments or the first argument is a primitive constant; this node is what would be returned by the `home-node` call on the first argument (see Section 3.8). These defaults can be adjusted using the `apply-default-local-node` directive in Chapter 4.

The J-Machine implementation also performs optimizations if *node* is a recognizable form such as `(local-node)`, in which case the function or method is more likely to be inlined.

Type Assertion

(:*type expression***)** **Primitive**

The type assertion statement asserts that the type of *expression*'s value is a subtype of *type*. It returns *expression*'s value. The compiler is not required to generate an error if *expression* evaluates to a value that is not a subtype of *type*, but it may do so.

Variable Bindings

(clet (*binding-spec****)** *body* **)** **Primitive**
(clet* (*binding-spec****)** *body* **)** **Macro**
binding-spec ::= *typed-opt-names* | **(***typed-opt-names* {*variable-declaration*} [*value*] **)**
typed-opt-names ::= *opt-name* (**,** *opt-name*)* [**:** *type*]
opt-name ::= *name* | **_**
variable-declaration ≡ <none currently defined>

`clet` creates local variable bindings and evaluates *body* within the scopes of those bindings. Each *name* specifies the name of a new variable. *Typed-opt-names* specifies one or more names separated by commas followed by an optional type. The character _ can be used to indicate an unnamed local variable; unnamed local variables can be used to evaluate the initial value expression without binding a name in the static scope. If *type* is not present, it defaults to `object`. If the long form of a *binding-spec* is used, the variables in *typed-opt-names* can be declared using declarations and can be given an initial value. *Value*, the initial value is an expression evaluated outside the scope of the `clet`. Each initial value is evaluated only once, even if it is assigned to more than one variable. The new variables are bound concurrently. Their initial values may be evaluated concurrently, and they are not guaranteed to be touched by the time *body* begins executing—in *body* the new variables may still contain cfutures.

The value returned by a `clet` is the value returned by the last statement in *body*.

`clet*` is the same as `clet` except that it brings the new variables into scope one at a time, so later variables can refer to earlier ones. As in `clet`, the new variables may be evaluated concurrently and are not guaranteed to be touched by the time *body* begins executing.

(let (*binding-spec****)** *body* **)** **Macro**
(let* (*binding-spec****)** *body* **)** **Macro**

`let` is the same as `clet` except that all newly-bound variables are touched before *body* begins executing. As with `clet`, the initial values are evaluated concurrently.

`let*` is the same as `let` except that it brings the new variables into scope one at a time, so later variables can refer to earlier ones. Unlike `let`, the new variables are evaluated sequentially.

(cset *name expression* **)** **Primitive**

`cset` sets the variable *name* to *expression*. The variable gets either the touched value of *expression* or a cfuture promising to evaluate *expression*. `cset` returns `nil`.

(set *name expression* **)** **Macro**

`set` sets the variable *name* to the value of *expression*. The value is touched before it is assigned to the variable, so the variable will not contain a cfuture or a future after this statement. `set` returns `nil`.

(cas *name comparison replacement* **)** **Primitive**
comparison ::= *expression*
replacement ::= *expression*

`cas` (compare-and-set) is an atomic[1] operation that checks whether the value of variable *name* is `eq` to the value of *comparison*. If so, the value of *replacement* is stored in variable *name* and `cas` returns `true`; otherwise, `cas` returns `false`. The value of variable *name* is never a cfuture when `cas` completes.

Multiple Values

The constructs below are used for receiving multiple values from methods and functions. Multiple values may not be supported by all implementations of Concurrent Smalltalk.

(mv-cset (*name****) (***funct arg****))** **Primitive**

`mv-cset` sets the variables *name** to the multiple values returned by (*funct arg**). The variables get either the touched return values of *expression* or cfutures promising to evaluate them. Some of the return values may be available before others. `mv-cset` returns `nil`.

(mv-set (*name****) (***funct arg****))** **Macro**

`mv-set` is just like `mv-cset` except that it touches all variables in *name** before continuing.

(mv-clet (*mv-binding-spec* ***) (***funct arg* ***)** *body***)** **Macro**
mv-binding-spec ::= *typed-opt-names* | **(***typed-opt-names* {*variable-declaration* }**)**
typed-opt-names ::= *opt-name* (**,** *opt-name*)* [**:** *type*]
opt-name ::= *name* | **_**
variable-declaration ≡

(mv-let (*mv-binding-spec* ***) (***funct arg* ***)** *body***)** **Macro**

`mv-clet` and `mv-let` are just like `clet` and `let` except that they initialize the new variables to the values returned by (*funct arg* *).

Syntactic Sugar

[*arg* ***]** **Macro**

This form is equivalent to (`get` *arg**).

[1] In the current implementation, in order for `cas` to be atomic, neither name nor replacement can be a future. If replacement could be a future, it should be forced before a `cas` is done. There is no easy solution if name could be a future. Fortunately, there is usually little reason to store a future in a semaphore.

(cset (*funct arg****) *expression***)** **Macro**

When the first argument of a `cset` is a function or a method call, `cset` is desugared into another function or a method call. The above forms are converted to (*funct' arg* * *expression*), where the identifier *funct'* is obtained by appending the characters `cput-` to the beginning of the identifier *funct*, unless:

funct is `get`, in which case *funct'* is `cput`;
funct is `get-`*x*, in which case *funct'* is `cput-`*x* (*x* is any sequence of characters);
funct is `put`, `put-`*x*, `cput`, `cput-`*x*, `cap`, or `cap-`*x*, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (*funct arg**) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

For example, `(cset (first sequence) 3)` is converted to `(cput-first sequence 3)`, while `(cset [big-array 7] 12)` is converted to `(cput big-array 7 12)`.

(set (*funct arg****) *expression***)** **Macro**

When the first argument of a `set` is a function or a method call, `set` is desugared into another function or a method call. The above forms are converted to (*funct' arg* * *expression*), where the identifier *funct'* is obtained by appending the characters `put-` to the beginning of the identifier *funct*, unless:

funct is `get`, in which case *funct'* is `put`;
funct is `get-`*x*, in which case *funct'* is `put-`*x* (*x* is any sequence of characters);
funct is `put`, `put-`*x*, `cput`, `cput-`*x*, `cap`, or `cap-`*x*, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (*funct arg**) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

For example, `(set (first sequence) 3)` is converted to `(put-first sequence 3)`, while `(set [big-array 7] 12)` is converted to `(put big-array 7 12)`.

(cas (*funct arg****) *comparison replacement* **)** **Macro**

When the first argument of a `cas` is a function or a method call, `cas` is desugared into another function or a method call. The above form is converted to (*funct' arg* * *comparison replacement*), where the identifier *funct'* is obtained by appending the characters `cap-` (compare-and-put) to the beginning of the identifier *funct*, unless:

funct is `get`, in which case *funct'* is `cap`;
funct is `get-`*x*, in which case *funct'* is `cap-`*x* (*x* is any sequence of characters);
funct is `put`, `put-`*x*, `cput`, `cput-`*x*, `cap`, or `cap-`*x*, in which case an error occurs.

Funct must be a function name or a method selector. It may not be an expression or a variable reference. (*funct arg**) may, however, be a macro or contain macros; these macros are expanded before the above conversion takes place.

Flow of Control

(begin *body*) **Primitive**

`begin` evaluates the statements in *body* sequentially, touching each one except the last before it begins the next, and returns the untouched value returned by the last one. If there are no statements in *body* , `begin` returns `nil`.

(nconcurrently *statement* *) **Macro**
(concurrently *statement* *) **Macro**
?*statement* **Macro**

These macros evaluate the statements in *statement* * concurrently and return `nil`. `concurrently` waits until all statements have finished executing before returning, while `nconcurrently` does not. `?`*statement* is an abbreviation for `(nconcurrently` *statement* `)`.

(if *test consequent* [*alternative*]**)** **Primitive**
test ::= *expression*
consequent ::= *expression*
alternative ::= *expression*

`if` evaluates the *test* expression, which must return either `true` or `false`. If it returns `true`, the *consequent* expression is evaluated and its value returned; otherwise, the *alternative* expression, if any, is evaluated and its value returned. `if` is not guaranteed to touch the *test* value. However, it is guaranteed to evaluate only the appropriate arm of the conditional.

(when *test body* **)** **Macro**
test ::= *expression*

`when` evaluates the *test* expression, which must return either `true` or `false`. If it is `true`, `when` evaluates *body* and returns its last value. If *test* evaluates to `false`, `when` returns `nil`.

(unless *test body* **)** **Macro**
test ::= *expression*

`unless` evaluates the *test* expression, which must return either `true` or `false`. If it is `false`, `unless` evaluates *body* and returns its last value. If *test* evaluates to `true`, `unless` returns `nil`.

(cond (*test body* **)*****)** **Macro**
test ::= *expression*

`cond` evaluates the *test* expressions, which must return either `true` or `false`, in order until it finds one that evaluates to `true`. That test's *body* is then evaluated and its value returned. If no *test* evaluates to true, `cond` returns `nil`.

Loops

Note: Optimist now generates true loops which run on a single node until completed or until they suspend for some reason such as a non-inlined function call or remote data access. Long, uninterrupted loops prevent other computations from taking place on the node and can cause network backups.

(while *test body* **)** **Primitive**
test ::= *expression*

`while` evaluates the *test* expression, which must return either `true` or `false`. As long as it returns `true`, *body* is evaluated and *test* reevaluated. When *test* evaluates to `false`, `while` returns `nil`.

(repeat *body* **until** *test* **)** **Primitive**
test ::= *expression*

`repeat` first evaluates *body* and then the *test* expression, which must return either `true` or `false`. As long as it returns `false`, `repeat` goes back to evaluating *body*. When *test* evaluates to `true`, `repeat` returns `nil`.

(for (*name start end***)** *body* **)** **Macro**
start ::= *expression*
end ::= *expression*

`for` evaluates *body* sequentially with variable *name* bound to consecutive integers starting from *start* inclusive and ending at *end* exclusive. If *end* is less than or equal to *start* , *body* is not evaluated. `for` returns `nil`. The values of *start* and *end* are evaluated before *body* begins executing.

Primitive Control

(block *continuation body* **)** **Macro**

`block` is just like begin except that it allows the use of `return` and `reply` statements to leave it. The statements in *body* are evaluated as in a `begin`. *Continuation* specifies the block's continuation for use in `return` and `reply` statements.

(_loop *body* **)** **Primitive**
(loop *continuation body* **)** **Macro**

`_loop` and `loop` define a loop body. The statements in *body* are evaluated as in a `begin`, except that after the last statement in *body* has been evaluated, the first statement is evaluated again, and so on. The loop does not terminate unless an explicit `return` or `reply` statement is encountered.

`_loop` generates a plain loop without defining a continuation; `loop` defines a loop with a continuation *continuation* which can be used in `return` and `reply` statements.

Returning Values

Since the last expression in the method code is implicitly returned to `continuation`, the statements below are necessary only if it is desired to return a value from the middle of a method or function, if a `block` or `loop` should be terminated, if multiple values are being returned, or if a value is returned to a continuation with a name other than `continuation`. `reply` and `exit` should be used with caution, as `exit` may cause the caller to hang, while `reply` may cause the caller to crash if two replies are inadvertently sent. *Care must be taken to reply to each continuation at most once*—sending a second reply to a continuation will almost certainly cause a system crash, and it is quite difficult to protect the system against this type of error. When using `reply` it is important to remember that there is an implicit `reply` of the last expression in the method code to `continuation`.

Continuations

Continuations are introduced by `lambda`, `method-lambda`, `defun`, `defmethod`, `block`, and `loop`. `lambda`, `method-lambda`, `defun`, and `defmethod` define the default continuation `continuation` unless told otherwise. They also reply to `continuation` if allowed to complete executing without an intervening `exit`. Thus, care must be taken when using nested function and method definitions to make sure that `reply` and `return` reply to the right continuation.

Continuation manipulation can become quite complicated, and not all features have to be supported by all implementations. A minimal implementation only has to allow replying to the innermost construct that defines continuations; hence, an implementation may restrict non-local replies. A more sophisticated implementation may allow replies to all continuations accessible in the current lexical scope. Finally, an advanced implementation may choose to make continuations first-class values of class `#:continuation` and allow them to be stored in variables.

(exit) **Primitive**

`exit` is a statement that hangs, never returning a value. In most cases `exit` can be thought of as exiting the current method or function, but it does not necessarily do so if used in a `cset`, `concurrently`, `nconcurrently`, `block`, or `loop` statement, `let` or `clet` bindings, or some other statement that permits parallel execution without synchronization.

(reply *expression***)** **Macro**
(reply (*continuation* *expression*)***)** **Primitive**

The first variant of `reply` evaluates *expression* and sends its value to `continuation`. Execution then proceeds with the next statement of the current method, if any. `reply` is not strict—it may reply a future or a cfuture. The value of a `reply` statement is `nil`.

The second variant of `reply` is used to return values to named continuations. The `reply` takes an even number of arguments; within each pair, the first argument is the continuation name and the second one its value.

(return *expression***)** **Macro**
(return (*continuation* *expression*)***)** **Macro**

`return` is equivalent to a `reply` followed by an `exit`—the values of the expressions are sent to the caller, and the execution of the method or function terminates subject to the caveats in the `exit` statement description.

(return-value-expected?) :boolean **Function**
(return-value-expected? *continuation***)** :boolean **Function**

`return-value-expected?` returns `true` if the caller of the method or function is expecting a reply for *continuation* (or `continuation` if *continuation* is not specified). It is not guaranteed to return `false` otherwise, so an implementation that always returns `true` is acceptable.

3.7. Built-in Methods and Functions

Built-in Classes

Table 3-2. Built-in Classes

Class	Metaclass	Values
`null`†	`primitive-class`	`nil`
`symbol`	`primitive-class`	Symbols, including `nil`, but not `true` and `false`
`true`†	`primitive-class`	The boolean `true`
`false`†	`primitive-class`	The boolean `false`
`boolean`	`primitive-class`	The booleans `true` and `false`
`character`	`primitive-class`	ASCII characters
`integer`	`primitive-class`	32-bit signed integers
`float`	`primitive-class`	Floating-point numbers [1]
`real`	`primitive-class`	Real numbers
`number`	`primitive-class`	Arbitrary numbers
`pair`	`standard-class`	Lisp-style cons cells
`primitive-class`	`primitive-class`	Primitive classes defined by Concurrent Smalltalk
`standard-class`	`primitive-class`	Standard (non-distributed) classes
`distributed-class`	`primitive-class`	Distributed classes
`class`†	`primitive-class`	General classes
`closure`	`primitive-class`	Closures
`function`	`primitive-class`	Functions, methods, and closures
`selector`	`primitive-class`	Method selectors
`funct`	`primitive-class`	Functions, methods, closures, and method selectors
`simple-stream`	`standard-class`	Streams for communicating with the outside world
`buffered-output-stream`	`standard-class`	Buffered output streams shadowing other streams
`stream`	`standard-class`	Sources of input or destinations for output
`simple-lock`	`primitive-class`	Very cheap and simple locks
`queueing-lock`	`primitive-class`	More expensive locks that queue pending tasks
`lock`	`standard-class`	General locks
`integer-array`	`primitive-class`	Small arrays of integers
`string`	`primitive-class`	Small arrays of characters
`boolean-array`	`primitive-class`	Small arrays of booleans
`packed-array`	`standard-class`	Small arrays of packed objects
`general-array`	`primitive-class`	Small arrays of arbitrary objects
`simple-array`	`standard-class`	Small arrays (confined to single processors)
`array`	`standard-class`	Arrays of arbitrary objects
`collection`	`standard-class`	Indexed collections of objects
`distobj`	`distributed-class`	All distributed objects
`physical-segment`	`primitive-class`	Cosmos physical segment descriptors
`_object-header`	`primitive-class`	Cosmos low-level object header words
`_context`	`primitive-class`	Used internally; no operations defined
`_continuation`	`standard-class`	Used internally; no operations defined
`object`	`standard-class`	All first-class values

Built-in classes are provided for reasons of efficiency and convenience. Many methods on built-in classes are compiled into single assembly language instructions instead of method calls, improving their speed greatly. Other built-in classes may be defined by methods writ-

---

†This class name conflicts with another global name, so it has to be preceded with `#:` whenever it is used.
[1]Floating point numbers may not be implemented in all Concurrent Smalltalk implementations.

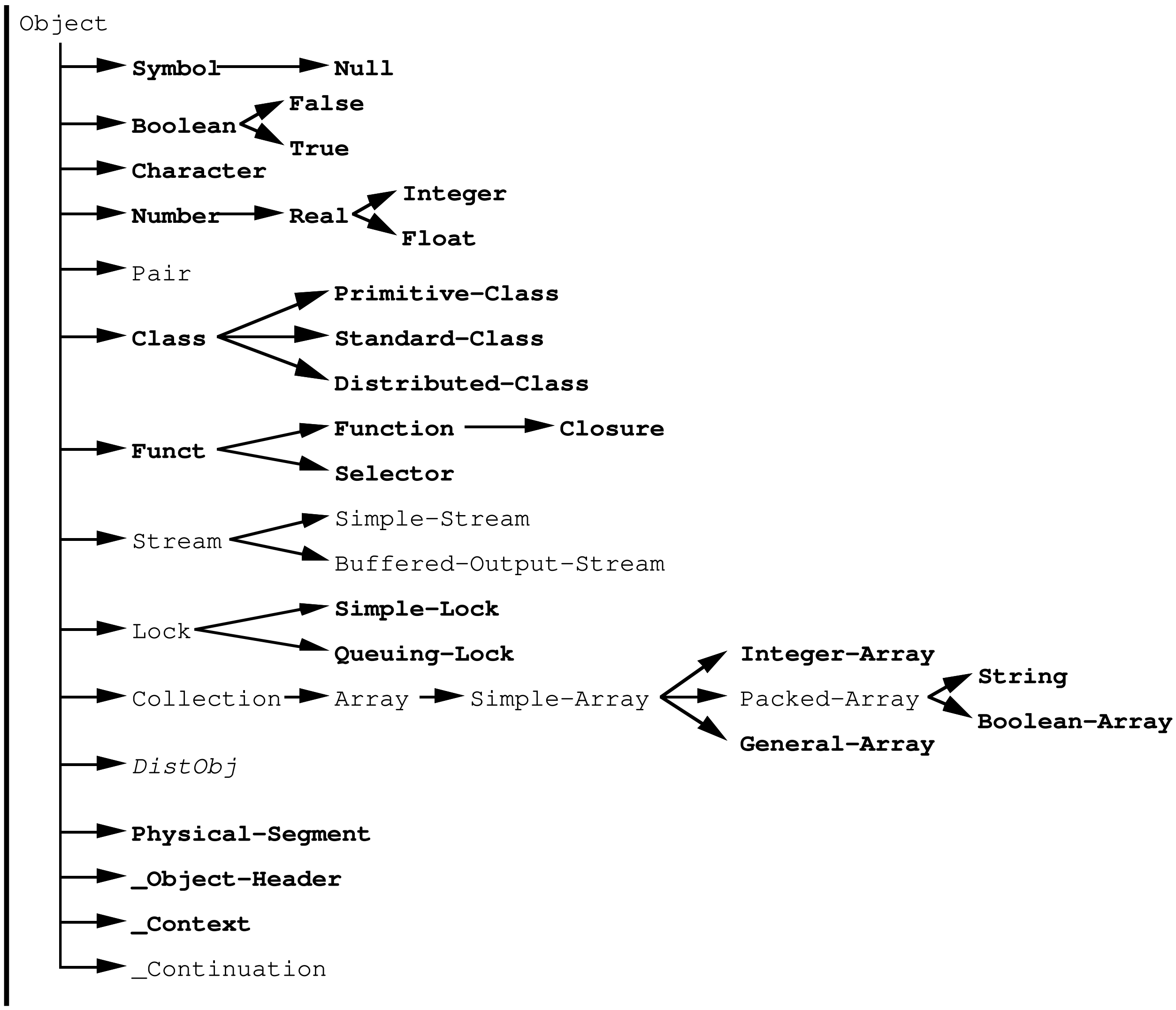

Figure 3-2. Hierarchy of built-in classes

The superclasses are shown to the left of their subclasses. All classes are subclasses of `object`. Classes with metaclass `primitive-class` are shown in bold, classes with metaclass `standard-class` are shown in standard type, and the one class with metaclass `distributed-class` is shown in italic. User-defined classes may be defined as subclasses of any of the classes having `standard-class` or `distributed-class` as a metaclass.

ten in assembly language and linked with the programs generated by the compiler. Arrays may be defined this way. The built-in classes are listed in Table 3-2, and their hierarchy is shown in Figure 3-2.

The *metaclass* of a class is the class of the class object itself. Metaclasses govern certain aspects of class behavior such as inheritance and the action of `new`. Only classes having `standard-class` or `distributed-class` as a metaclass permit user-defined subclasses. At the implementation's discretion some classes with `primitive-class` as a metaclass may actually be instances of `standard-class` or `distributed-class`, but portable programs should not rely on this.

Primitive Selectors

Built-in methods are provided for the basic arithmetic and logical operations. The methods are explained in the following sections. Some built-in method calls compile into assembly language instructions. The selectors and methods used for those calls are called primitive (restricted in [3]), and user-defined methods cannot be defined for them. Moreover, the `method` directive cannot be used on primitive selectors to obtain methods. These selectors are listed in Table 3-3 and are also labeled in their descriptions.

Table 3-3. Primitive Selectors

```
not and or xor lognot logand logor logxor
< <= > >= = <>
neg + - * / // mod
ash integer-length
```

3.8. Objects

Objects

(new *c*:standard-class**)** :object **Macro**
(local-new *c*:standard-class**)** :object **Macro**
(new-on *node* :integer *c*:standard-class**)** :object **Macro**
(new @*node* *c*:standard-class**)** :object **Macro**

`new`, when applied to a standard class, creates and returns a new instance object of the specified class on a random node. The object's data is not initialized.

`local-new` is the same as `new` except that the object is created on the local node. `new-on` creates the object on the specified node (see node numbers later in this section). The node number must be valid. For compatibility with older versions of Concurrent Smalltalk, *node* can also be one of the symbols (not expressions) `'here`, `'west`, `'east`, `'south`, `'north`, `'down`, `'up`, `'neighbor`, or `'random`, which are shorthands for `(local-node)`, `(direction-node west-direction)`, ..., `(neighbor-node)`, and `(random-node)`. However, the use of these shorthands is discouraged for future compatibility.

The last form, `(new @`*node* *c*`)`, is just an alternate syntax for `new-on`.

`new`, `local-new`, and `new-on` can only allocate objects of standard classes of fixed-size objects. For lower-level primitives for allocating relocatable objects, see `new-size`, `local-new-size`, and `new-on-size` in Section 3.16.

Copiers

(deep-copy *o*:object**)** :object **Method**

`deep-copy` returns a copy of the object. Any of the object's instance variables are also recursively copied using `deep-copy`. If the class of the object is immutable, `deep-copy` may just return the object it received. `deep-copy` may fail to terminate on circular object references.

(shallow-copy *o*:object**)** :object **Method**

`shallow-copy` returns a copy of the object without copying any of the object's instance variables. If the class of the object is immutable, `shallow-copy` may just return the object it received.

(copy *o*:object**)** :object **Method**

`copy` is the most appropriate copying routine for a given object. It defaults to `shallow-copy`.

Deallocators

In addition to waiting for garbage collection, the following methods can be used to explicitly deallocate the storage for an object. Accessing an object after it has been deallocated causes an error.

(deep-dispose *o*:object**)** :null **Method**

`deep-dispose` deallocates the object's storage. Any of the object's instance variables are also recursively disposed using `deep-dispose`. `deep-dispose` should not be used on circular or multiple object references. Also, `deep-dispose` should not be called on a distributed object because it will just dispose one constituent, corrupting the distributed object structure.

(shallow-dispose *o*:object**)** :null **Method**

`shallow-dispose` deallocates the object's storage without disposing any of the object's instance variables. `shallow-dispose` should be used with caution on distributed objects, as it will only dispose one constituent; take care never to refer to the distributed object again after one or more of its constituents are deallocated.

(dispose *o*:object**)** :null **Method**

`dispose` is the most appropriate deallocating routine for a given object. It defaults to `shallow-dispose`. `dispose` should be used with caution on distributed objects, as it will only dispose one constituent; take care never to refer to the distributed object again after one or more of its constituents are deallocated.

Class Inquiries

(symbol? *o*:object**)** :boolean **Function**
(null? *o*:object**)** :boolean **Function**
(false? *o*:object**)** :boolean **Function**
(true? *o*:object**)** :boolean **Function**
(boolean? *o*:object**)** :boolean **Function**
(character? *o*:object**)** :boolean **Function**
(integer? *o*:object**)** :boolean **Function**
(float? *o*:object**)** :boolean **Function**
(real? *o*:object**)** :boolean **Function**
(number? *o*:object**)** :boolean **Function**
(primitive-class? *o*:object**)** :boolean **Function**
(standard-class? *o*:object**)** :boolean **Function**
(distributed-class? *o*:object**)** :boolean **Function**
(class? *o*:object**)** :boolean **Function**
(closure? *o*:object**)** :boolean **Function**
(function? *o*:object**)** :boolean **Function**
(selector? *o*:object**)** :boolean **Function**
(funct? *o*:object**)** :boolean **Function**
(integer-array? *o*:object**)** :boolean **Function**
(string? *o*:object**)** :boolean **Function**
(boolean-array? *o*:object**)** :boolean **Function**
(packed-array? *o*:object**)** :boolean **Function**
(general-array? *o*:object**)** :boolean **Function**
(simple-array? *o*:object**)** :boolean **Function**
(array? *o*:object**)** :boolean **Function**
(collection? *o*:object**)** :boolean **Function**
(distobj? *o*:object**)** :boolean **Function**
(physical-segment? *o*:object**)** :boolean **Function**
(_object-header? *o*:object**)** :boolean **Function**

These predicates return `true` if the object is a member of the corresponding class or its descendants. Note that a few built-in classes don't have predicates, including `object`, `_context`, and `_continuation`.

(class-kind? *o*:object *c*:class**)** :boolean **Method**
(class-member? *o*:object *c*:class**)** :boolean **Method**

`class-kind?` returns a boolean value that specifies whether the given object is an instance of the given class or one of its subclasses. `class-member?` is just like `class-kind?` except that it returns `true` only if the object is a direct instance of the given class.

(class-of *o*:object**)** :class **Primitive**

`class-of`, when applied to an object, returns its class.

(class-name *o*:object**)** :string **Method**

`class-name`, when applied to an object, returns the name of its class as a string. The name is derived from the class name in the source code by capitalizing it.

(subclass? *c1* , *c2* :class**)** :boolean **Primitive**

`subclass?` returns `true` if *c1* is a subclass of *c2* and `false` otherwise.

Node Numbers

Machine node numbers (NNRs) are represented as integers for efficiency. The mapping from (x, y, z) node coordinates to node numbers is implementation-dependent. For the J-Machine implementation, node numbers are NNR register values, which means that node numbers are not contiguous.

The following operations are provided to manipulate node numbers. All of the functions are inlined for efficiency by default.

max-x-node:integer **Constant**
max-y-node:integer **Constant**
max-z-node:integer **Constant**

These constants contain the maximum *x*, *y*, and *z* extents of the block of node numbers available to the current program. The allowable *x* coordinates range from 0 to `max-x-node`, inclusive, and likewise for *y* and *z* coordinates.

(make-nnr *x*, *y*, *z* :integer**)** :integer **Function**

`make-nnr` returns the node number matching the given coordinates. The *x*, *y*, and *z* coordinates should be within range for the currently allocated J-Machine block; no range check or wraparound is done.

(nnr-x *node* :integer**)** :integer **Function**
(nnr-y *node* :integer**)** :integer **Function**
(nnr-z *node* :integer**)** :integer **Function**

`nnr-x`, `nnr-y`, and `nnr-z` extract the coordinates from the given node number.

_node-mask:integer **Constant**
(_make-safe-nnr *node* :integer**)** :integer **Function**

The `_node-mask` constant is the node number of the node with the maximum *x*, *y*, and *z* coordinates. `_make-safe-nnr` masks any integer to become a valid node number.

(local-node) :integer **Primitive**

`local-node` returns the node number of the node on which this statement is executed.

(random-node) :integer **Primitive**

`random-node` returns the node number of a random node on the machine.

west-direction:integer **Constant**
east-direction:integer **Constant**
south-direction:integer **Constant**
north-direction:integer **Constant**
down-direction:integer **Constant**
up-direction:integer **Constant**
(direction-node direction:integer**)** :integer **Macro**

`direction-node` returns the node number of the neighboring node in the given direction. The mesh of nodes is considered to be a torus, so the `direction-node` relation wraps around the edges of the mesh. *Direction* must be one of the six integer constants above.

(neighbor-node) :integer **Primitive**

`neighbor-node` returns the node number of a random neighboring node (one of the six neighbors returned by `direction-node`).

(home-node *o*:object**)** :integer **Primitive**

`home-node` returns the node number of the node containing the given object. If the object is a distributed object, the node containing the nearest constituent is returned. If the object is a one-word primitive object not allocated on a heap, `home-node` returns a random node number. The home node of an object is where a function or method call on that object is generally executed.

3.9. Distributed Objects

distobj **Class**

`distobj` is the distributed object class.

Group and Constituents

A distributed object consists of a *group* name and one or more *constituent objects*. The constituent objects act just like normal objects except that they inherit methods and instance variables from the class `distobj` and they respond to the `group` and `get-group` messages. A group name indicates the entire collection of distributed objects. When a method is called on the group name, it is processed by one of the *distobj*'s constituent objects, as though the method were called on that constituent object. The identity of the constituent object receiving the message is left unspecified; implementations are encouraged to heuristically pick different constituent objects for different calls to the group, thereby facilitating concurrency for distributed object operations. When a constituent object is processing a method, `self` is the constituent object, not the group name.

Creation

(new *c*:distributed-class *n-constituents* :integer**)** :distobj **Macro**

`new`, when applied to a distributed class, creates and returns a new distributed object of the specified class with the given logical number of constituents. The constituents are not initialized.

The distributed object that is created may contain more constituents than *n-constituents*. The runtime system determines an appropriate physical number of constituents for the distributed object that is at least as large as *n-constituents*; the current implementation rounds up *n-constituents* to the nearest power of 2. The additional constituents should be prepared to respond to messages sent to the distributed object.

To allocate distributed objects with constituents of a parametrized size, see `new-size` in Section 3.16.

Operations

(co *o*:distobj *n*:integer**)** :distobj **Primitive**

`co` returns the *n*th constituent object of the distributed object. *O* can be either the group object or any of its constituents. *N* must be between 0 and the physical number of constituent objects in the *distobj* minus one.

(logical-limit *o*:distobj**)** :integer **Method**

`logical-limit` is the logical number of constituent objects in the distributed object.

(physical-limit *o*:distobj**)** :integer **Method**

`physical-limit` is the physical number of constituent objects in the distributed object. The constituent objects are numbered between 0 and `physical-limit` minus one, inclusive. `physical-limit` is never less than `logical-limit`.

(index *o*:distobj**)** :integer **Method**

`index` is the number of a particular constituent object in a distributed object. `index` ranges between 0 and `physical-limit` minus one, inclusive.

group:distobj **Instance Variable**
(group *o*:distobj**)** :distobj **Method**
(get-group *o*:distobj**)** :distobj

`group` is the inverse of `co`—it returns the group object of the given distributed object. *O* can be either the group object or any of its constituents; if *o* is already a group object, `group` just returns it. `get-group` is functionally equivalent to `group`; it is provided to avoid name conflicts with the `group` variable inside distributed object methods.

3.10. Logical and Arithmetic Operations

Comparisons

(eq *o1*, *o2* :object**)** :boolean — **Primitive**
(neq *o1*, *o2* :object**)** :boolean — **Primitive**

`eq` returns `true` if the two objects are indistinguishable—there is no legal way of distinguishing *o1* from *o2*. For mutable objects this means that *o1* and *o2* are the same object. For immutable objects, `eq` may in addition return `true` if *o1* and *o2* are different objects that contain the same data.

`neq` is the logical negation of `eq`.

(= *o1*, *o2* :number**)** :boolean — **Primitive Method**
(<> *o1*, *o2* :number**)** :boolean — **Primitive Method**

(= *o1*, *o2* :boolean**)** :boolean — **Primitive Method**
(<> *o1*, *o2* :boolean**)** :boolean — **Primitive Method**

(= *o1*, *o2* :symbol**)** :boolean — **Primitive Method**
(<> *o1*, *o2* :symbol**)** :boolean — **Primitive Method**

(= *o1*, *o2* :character**)** :boolean — **Primitive Method**
(<> *o1*, *o2* :character**)** :boolean — **Primitive Method**

These comparisons return `true` if *o1* is equal to or not equal to *o2*, respectively. Equality means numeric equality for numbers; an `integer` is considered equal to a mathematically equal `float`.

(< *m1*, *m2*:number **)** :boolean — **Primitive Method**
(<= *m1*, *m2*:number **)** :boolean — **Primitive Method**
(> *m1*, *m2*:number **)** :boolean — **Primitive Method**
(>= *m1*, *m2*:number **)** :boolean — **Primitive Method**

(< *m1*, *m2*:boolean **)** :boolean — **Primitive Method**
(<= *m1*, *m2*:boolean **)** :boolean — **Primitive Method**
(> *m1*, *m2*:boolean **)** :boolean — **Primitive Method**
(>= *m1*, *m2*:boolean **)** :boolean — **Primitive Method**

These comparisons return `true` if *m1* is less than *m2*, *m1* is less than or equal to *m2*, *m1* is greater than *m2*, or *m1* is greater than or equal to *m2*, respectively. For the purposes of comparison, `false` is considered to be less than `true`.

Note that these comparisons will not work on `characters`. Use `char<`, `char<=`, `char>`, and `char>=` to compare `characters`.

Logical Operations

(not *b* :boolean**)** :boolean — **Primitive Method**

`not` returns the logical negation of *b*.

(and (*b* :boolean)* **)** :boolean — **Primitive Method**

`and` returns the logical AND of its arguments. If no arguments are specified, `and` returns `true`.

(or (*b* :boolean)* **)** :boolean **Primitive Method**

`or` returns the logical inclusive OR of its arguments. If no arguments are specified, `or` returns `false`.

(xor (*b* :boolean)* **)** :boolean **Primitive Method**

`xor` returns the logical exclusive OR of its arguments. If no arguments are specified, `xor` returns `false`.

(sc-and (*b* :boolean)* **)** :boolean **Macro**
(sc-or (*b* :boolean)* **)** :boolean **Macro**

These are short-circuit versions of `and` and `or`. They evaluate arguments sequentially from left to right only as far as necessary for the answer to be unambiguously determined.

Bitwise Logical Operations

(lognot *b* :boolean**)** :boolean **Primitive Method**
(logand (*b* :boolean)+ **)** :boolean **Primitive Method**
(logor (*b* :boolean)+ **)** :boolean **Primitive Method**
(logxor (*b* :boolean)+ **)** :boolean **Primitive Method**

(lognot *b* :integer**)** :integer **Primitive Method**
(logand (*b* :integer)* **)** :integer **Primitive Method**
(logor (*b* :integer)* **)** :integer **Primitive Method**
(logxor (*b* :integer)* **)** :integer **Primitive Method**

These methods perform bitwise logical operations. When called on `booleans`, they perform the same operations as `not`, `and`, `or`, and `xor`, respectively. When called on `integers`, they perform the corresponding operations bitwise on semi-infinite two's complement representations of the integers, treating 0 as `false` and 1 as `true`. The integers do not have to be internally stored in the two's complement form; all that is necessary is that these operations act as if they were. When supplied with no arguments, `logand` returns -1, while `logor` and `logxor` return 0.

Arithmetic Operations

For most binary arithmetic operations, the class of the result is the class of the most general argument. For example, if two `integers` are added, the result is an `integer`, but if an `integer` and a `float` are added, the result is a `float`. User-defined classes may define other numeric subclasses, in which case they have to handle appropriate coercions themselves—if a `number` is added to a member of a user-defined subclass of `number`, the + method for the user-defined subclass will have to dispatch on the type of its second argument.

Note : When using arithmetic operations, especially floating-point ones, the code generated will be much more efficient if the compiler knows the types of arguments. Specifically, `neg`, `+`, `-`, `*`, comparisons, and `ash` will be compiled to direct floating point library calls rather than causing MDP exceptions if the compiler knows that they are applied to `floats` and the `optimize-float-library-calls` option is set; there is no need to declare types if these are used only on `integers`. On the other hand, `abs` and `sqrt` will be compiled more efficiently, avoiding a type check, if the compiler can deduce that they will be passed only `floats` or only `integers`.

(zero? *n*:number **)** :boolean **Method**

`zero?` returns `true` if *n* is zero and `false` otherwise.

(plus? *n*:number **)** :boolean **Method**

`plus?` returns `true` if *n* is greater than zero and `false` otherwise.

(minus? *n*:number **)** :boolean **Method**

`minus?` returns `true` if *n* is less than zero and `false` otherwise.

(abs *n*:number **)** :number **Method**

`abs` returns the absolute value of *n*. The class of the result value is the same as the class of *n*.

(max *n1* , *n2* :number**)** :number **Method**
(min *n1* , *n2* :number**)** :number **Method**

`max` returns the greater of *n1* and *n2* , while `min` returns the lesser one.

(neg *n*:number **)** :number **Primitive Abstract Method**

`neg` returns the negation of *n*. The class of the result value is the same as the class of *n*.

(+ (*n*:number)***)** :number **Primitive Abstract Method**

+ returns the sum of its arguments. If no arguments are specified, + returns 0.

(– *n1* , *n2* :number**)** :number **Primitive Abstract Method**

– returns the difference of its arguments, *n1* -*n2* .

(* (*n*:number)***)** :number **Primitive Abstract Method**

`*` returns the product of its arguments. If no arguments are specified, `*` returns 1.

(/ *n1* , *n2* :number**)** :number **Primitive Abstract Method**

/ returns the quotient of its arguments, *n1*/ *n2* . If *n1* and *n2* are both `integers` and *n1* is not exactly divisible by *n2* , the result is a `float`. If *n2* is zero, either an error occurs or some representation of infinity is substituted as an answer.

(// *n1* , *n2* :integer**)** :integer **Primitive Method**

`//` returns the integer quotient of its arguments rounded towards -∞, $\lfloor n1/n2 \rfloor$. If *n2* is zero, either an error occurs or some representation of infinity is substituted as an answer. Having `//` round towards -∞ allows the use of `ash` to divide when the divisor is an integral power of two.

(mod *n1* , *n2* :integer**)** :integer **Primitive Method**

`mod` returns the nonnegative remainder of dividing *n1* by *n2* , $n1 - n2 * \lfloor n1/n2 \rfloor$. If *n2* is zero, either an error occurs or some representation of an indeterminate number is substituted as an answer. Having `mod` return the nonnegative remainder allows the use of `logand` to find the remainder when the divisor is an integral power of two. When the remainder is nonzero, its sign is always the same as the sign of the divisor *n2* . Also, `(+ (mod` *n1* *n2* `) (*` *n2* `(//` *n1* *n2*`)))` ≡ *n1* .

(divmod *n1* , *n2* :integer**)** :: **(***quotient*:integer *remainder* :integer**)** **Primitive**

`divmod` returns both the integer quotient, as computed by `//`, and the remainder, as computed by `mod`, of *n1* and *n2* . Using `divmod` is more efficient than calling `//` and `mod` separately.

(ash *n1* :integer *n2* :integer**)** :integer **Primitive Method**
(ash *n1* :float *n2* :integer**)** :float **Primitive Method**

`ash` returns *n1* multiplied by two raised to the `n2`th power, $n1 * 2^{n2}$. If *n1* is a `float`, no rounding takes place—the operation is simply an exponent adjustment. However, if *n1* is an `integer` and *n2* is negative, the result is rounded towards -∞.

See also `_lsh` and `_rot` in Section 3.16.

(integer-length *n*:integer**)** :integer **Primitive Method**

`integer-length` returns the bit "size" of *n*. For positive *n* this is $\lceil \log_2(n+1) \rceil$, while for negative *n* it is equal to $\lceil \log_2(-n) \rceil$.

Floating-Point Operations

(integer-to-fp *n*:integer**)** :float **Primitive**

`integer-to-fp` converts an `integer` to a `float`.

(sqrt *n*:number **)** :float **Abstract Method**

`sqrt` returns the nonnegative square root of *n*. The result is always a float, even if *n* is an integer.

(floor *n*:float **)** :integer **Primitive**

`floor` returns the greatest integer less than or equal to *n*.

Character Operations

(char-code *c*:character**)** :integer **Function**

`char-code` returns ASCII code of character *c*.

(code-char *n*:integer**)** :character **Function**

`code-char` returns a character with ASCII code *n*, which must be an integer between 0 and 255.

(char< *c1* , *c2* :character**)** :boolean **Function**
(char<= *c1* , *c2* :character**)** :boolean **Function**
(char> *c1* , *c2* :character**)** :boolean **Function**
(char>= *c1* , *c2* :character**)** :boolean **Function**

These comparisons compare the ASCII codes of *c1* and *c2* and return `true` if *c1* is less than *c2* , *c1* is less than or equal to *c2* , *c1* is greater than *c2* , or *c1* is greater than or equal to *c2* , respectively. Case is significant.

Use = and <> to compare characters for equality.

3.11. Lists

pair:standard-class **Class**

The Concurrent Smalltalk library provides a few functions that manipulate Lisp-style lists, which are chains of objects of class `pair` terminated by `nil`. Each `pair` object has two instance variables, `car` and `cdr`, which can be read and written normally. All of the standard accessor methods are defined, the main ones of which are documented below.

(car *l*:pair**)** :object **Method**
(get-car *l*:pair**)** :object **Method**
(put-car *l*:pair *value* :object**)** :pair **Method**

These functions return and set the value of the `car` instance variable of the `pair` object. Note that, unlike in Common Lisp, calling `car` on `nil` is an error.

(cdr *l*:pair**)** :object **Method**
(get-cdr *l*:pair**)** :object **Method**
(put-cdr *l*:pair *value* :object**)** :pair **Method**

These functions return and set the value of the `cdr` instance variable of the `pair` object. Note that, unlike in Common Lisp, calling `cdr` on `nil` is an error.

(length *l*:null**)** :integer **Method**
(length *l*:pair**)** :integer **Method**

`length` returns the length of the list, following `cdr` pointers. The length of `nil` is 0.

Creating

Lists can be created using `quote`, `new`, and `local-new`. In addition, the `cons` function also creates lists.

(cons *car*, *cdr* :object**)** :pair **Function**

`cons` creates and returns a new `pair` with the given `car` and `cdr` values. The `pair` is created on the local node.

3.12. Strings and Arrays

array:standard-class **Class**
simple-array:standard-class **Class**
general-array:primitive-class **Class**
integer-array:primitive-class **Class**
packed-array:standard-class **Class**
string:primitive-class **Class**
boolean-array:primitive-class **Class**

Strings and arrays are the primitive data structures for keeping track of indexed collections of data. All primitive strings and arrays are subclasses of the class `array`. `strings`, and `boolean-arrays` are packed to conserve space and time. `string` and `boolean-array` are subclasses of `packed-array`, which is a subclass of `array`.

Creating Arrays

(new-general-array *size* :integer**)** :general-array **Function**
(new-general-array-on *node* :integer *size* :integer**)** :general-array **Function**
(new-general-array @*node* *size*:integer **)** :general-array **Macro**

`new-general-array` creates a new `general-array` of arbitrary objects on the local node. *Size* specifies the number of elements in the array; the elements are numbered 0 through *size* -1. The array's elements are not initialized. `new-general-array-on` creates the array on the node with the given number, which must be a valid node number. The last form is just an alternate syntax for `new-general-array-on`.

(new-integer-array *size* :integer**)** :integer-array **Function**
(new-integer-array-on *node* :integer *size* :integer**)** :integer-array **Function**
(new-integer-array @*node* *size*:integer **)** :integer-array **Macro**

`new-integer-array` creates a new `integer-array` of integers on the local node. *Size* specifies the number of elements in the array; the elements are numbered 0 through *size* -1. The array's elements are not initialized. `new-integer-array-on` creates the array on the node with the given number, which must be a valid node number. The last form is just an alternate syntax for `new-integer-array-on`.

(new-string *size* :integer**)** :string **Function**
(new-string-on *node* :integer *size* :integer**)** :string **Function**
(new-string @*node* *size*:integer **)** :string **Macro**

`new-string` creates a new array of characters, also called a `string`, on the local node. *Size* specifies the number of el ements in the array; the elements are numbered 0 through *size* -1. The array's ele ments are not initialized. `new-string-on` creates the string on the node with the given number, which must be a valid node number. The last form is just an alternate syntax for `new-string-on`.

(new-boolean-array *size* :integer**)** :boolean-array **Function**
(new-boolean-array-on *node* :integer *size* :integer**)** :boolean-array **Function**
(new-boolean-array @*node* *size*:integer **)** :boolean-array **Macro**

`new-boolean-array` creates a new `boolean-array` of booleans on the local node. *Size* specifies the number of elements in the array; the elements are numbered 0 through *size* -1. The array's elements are not initialized. `new-boolean-array-on` creates the array on the node with the given number, which must be a valid node number. The last form is just an alternate syntax for `new-boolean-array-on`.

Operations on Entire Arrays

The following functions operate on entire arrays. Currently they call their function parameter *f* sequentially, but the implementation can be optimized or overridden to call *f* in concurrently if desired.

(fill *a* :array *value* :object**)** :array **Abstract Method**

`fill` destructively writes *value* to every element of the given array. If the array is an `integer-array`, a `string`, or a `boolean-array`, the value must have the correct type or the results are unspecified. `fill` returns the updated array.

(fill-range *a* :array *begin* , *end*:integer *value*:object **)** :array **Abstract Method**

`fill-range` destructively writes *value* to every array element with index between *begin* inclusive and *end* exclusive. If *end* is less than or equal to *begin*, nothing happens; otherwise, *begin* and *end* should be such that only elements in the array are filled. If the array is an `integer-array`, a `string`, or a `boolean-array`, the value must have the correct type or the results are unspecified. `fill` returns the updated array.

(init *a* :array *f*:funct**)** :array **Abstract Method**

`init` calls *f* on integers between 0 and the size of *a* minus one, inclusive, and stores the results in the corresponding elements of *a* . *f* should not access the values of elements of *a*. `init` returns the *a* array after all calls to *f* have returned.

(map *src*:array *dst*:array *f* :funct**)** :array **Abstract Method**

`map` calls *f* on each element of the *src* array and stores the results in the corresponding elements of the *dst* array. The sizes of the two arrays must be equal. If *src* is a `simple-array`, so must be *dst*. *Src* and *dst* may be the same array. *f* should not access the values of elements of *dst*. `map` returns the *dst* array after all calls to *f* have returned.

(for-each *a* :array *f*:funct**)** :array **Abstract Method**
(nfor-each *a* :array *f*:funct**)** :array **Abstract Method**

Both of the above methods call *f* on each element of the array and then return the array without modifying it. `nfor-each` does not wait until any of the calls to *f* return, while `for-each` does.

Accessing Arrays

[*a* :array *pos*:integer **]** :object **Abstract Method**
(get *a* :array *pos*:integer **)** :object

`get` returns the element at position *pos* of the given array. `get` may signal an error if *pos* is outside the bounds of the array. The results of accessing an uninitialized element are unspec ified.

(set [*a* :array *pos*:integer **]** *value* :object**)** :array **Abstract Method**
(put *a* :array *pos*:integer *value* :object**)** :array

`put` destructively writes *value* at position *pos* of the given array. `put` may signal an error if *pos* is outside the bounds of the array. If the array is an `integer-array`, a `string`, or a `boolean-array`, *value* must have the correct type or the results are unspecified. `put` returns the updated array.

(size *a* :array**)** :integer **Abstract Method**

`size` returns the number of elements in the array, as specified when the array was created.

Strings

(string-append string*) :string **Macro**

`string-append` appends all of the strings and returns the combined string. `string-append` returns the empty string if no strings are given.

(string= *s1* , *s2* :string**)** :boolean **Function**
(string<> *s1* , *s2* :string**)** :boolean **Function**
(string< *s1* , *s2* :string**)** :boolean **Function**
(string<= *s1* , *s2* :string**)** :boolean **Function**
(string> *s1* , *s2* :string**)** :boolean **Function**
(string>= *s1* , *s2* :string**)** :boolean **Function**

These comparisons compare two strings and return `true` if *s1* is equal to *s2* , *s1* is not equal to *s2* , *s1* is less than *s2* , *s1* is less than or equal to *s2* , *s1* is greater than *s2* , or *s1* is greater than or equal to *s2* , respectively. Case is significant.

(strcmp *s1* , *s2* :string**)** :integer **Function**

`strcmp` returns 1 if *s1* is greater than *s2* , 0 if they are equal, and -1 if *s1* is less than *s2* . Case is significant.

3.13. Locks

Locks are used to synchronize processes. A lock can be *acquired* by only one process at a time, and the acquiring operation is atomic. After a process has acquired a lock, it can proceed to perform whatever exclusive operations it wants to do. When it is done, it should *release* the lock to make it available again. If a process attempts to acquire a lock that is busy (acquired), it will wait until the lock is available.

`lock`:standard-class **Class**
`simple-lock`:primitive-class **Class**
`queueing-lock`:primitive-class **Class**

Two built-in lock classes are provided: `simple-lock` and `queueing-lock`. `simple-lock` is a very cheap and fast implementation intended for situations in which a lock is not acquired for long periods of time and there is little contention for the lock. `simple-locks` are adequate for most purposes. `queueing-locks` are heavy-duty locks for use in situations where there may be significant contention for a lock.

Lock Operations

`(new-simple-lock)`:simple-lock **Function**
`(new-queueing-lock)`:queueing-lock **Function**

`new-simple-lock` creates a new simple lock, while `new-queueing-lock` creates a new queueing lock. The lock is initially available.

`(init` *l*:lock **`)`**:null **Abstract Method**

`init` reinitializes the lock, making it available regardless of its previous state.

`(acquire` *l*:lock **`)`**:null **Abstract Method**

`acquire` acquires the lock. If the lock is busy, `acquire` waits until the lock is available before acquiring it and returning.

`(release` *l*:lock **`)`**:null **Abstract Method**

`release` releases the lock. If the lock is already available, `release` signals an error.

`(busy?` *l*:lock **`)`**:boolean **Abstract Method**

`busy?` returns `true` if the lock is busy and `false` otherwise.

`(with-locks ((`*l* :lock)* **`)`** body**`)`** **Macro**

`with-locks` first acquires all of the locks listed, in the order in which they are listed, then evaluates *body*, and finally releases all of the locks. It returns the value of *body*.

3.14. Input and Output

Streams

stream:standard-class **Class**
simple-stream:standard-class **Class**
buffered-output-stream:standard-class **Class**

Streams are sources and sinks of data. A stream is usually a connection to a terminal or to a file, but other uses of streams are possible. Concurrent Smalltalk defines a general class `stream` as well as specific implementations of streams, `simple-stream`, and `buffered-output-stream`. A `simple-stream` is used for sending output to an external device or file. A `buffered-output-stream` buffers information sent to another stream and forwards it to the stream in larger chunks; characters are forwarded either when the buffer is filled, a new-line character is printed, or when `force-output` is called. Other user-defined stream classes may be defined as subclasses of `stream`.

Input and Output Streams

unbuffered-terminal-stream:simple-stream **Global**
terminal-stream:buffered-output-stream **Global**

`unbuffered-terminal-stream` is the `simple-stream` used for interaction with the terminal. `terminal-stream` is a buffered stream for output to the terminal; using it is more efficient than using `unbuffered-terminal-stream`.

See also Terminal Stream Operations at the end of this section.

Creating Streams

(new-simple-stream *port-number* :integer**)** :simple-stream **Function**

`new-simple-stream` creates and returns a new `simple-stream` that sends its data to the given port number. Port number 1 is used for general text output. Other ports may be defined by Cosmos or the low-level firmware that communicates with the J-Machine.

(get-port-number *s*:simple-stream **)** :integer **Method**

`get-port-number` returns the `simple-stream`'s port number.

default-stream-buffer-length:integer **Constant**
(new-buffered-output-stream *s*:stream *buffer-length*:integer **)** :buffered-output-stream **Method**

`new-buffered-output-stream` creates and returns a new `buffered-output-stream` that forwards its output to stream *s*. *buffer-length* specifies the length of the buffer. `default-stream-buffer-length` is a good default value for *buffer-length*.

(get-underlying-stream *s*:buffered-output-stream **)** :stream **Method**

`get-underlying-stream` returns the stream to which the `buffered-output-stream` is forwarding its data.

(get-buffer-length *s*:buffered-output-stream **)** :integer **Method**

`get-buffer-length` returns the size of the `buffered-output-stream`'s buffer.

Reading

Currently no operations are defined for input from streams. See [3] for a proposed implementation of input.

Writing

The results of stream-writing and formatting methods and macros are not specified. They may be `null`, or may return some other value.

(write-stream-char *s*:stream *ch*:character **)** **Abstract Method**

`write-stream-char` writes character *ch* onto stream *s* .

(write-stream-string *s*:stream *string*:string**)** **Method**

`write-stream-string` writes *string* onto stream *s*. `write-stream-string` is equivalent to calling `write-stream-char` on each character in *string*.

(write-stream *s*:stream *o*:object**)** **Method**

`write-stream` writes some representation of the given Concurrent Smalltalk objects onto stream *s* . It is the same as `write` with the arguments transposed. Care should be taken when writing circular structures to make sure that `write-stream` terminates.

(force-output *s*:stream **)** **Method**

`force-output` flushes the output cached in the stream. Currently this method does nothing for `simple-streams` while flushing the buffer from `buffered-output-streams`.

Formatting

(terpri *s*:stream **)** **Method**

`terpri` writes a newline onto stream *s*.

(write-space *s*:stream **)** **Macro**

`write-space` writes a space onto stream *s*.

(write *o*:object *s* :stream**)** **Abstract Method**

`write` is used for formatting arbitrary objects for the purposes of `write-stream`. `write` should output some readable representation of object *o* onto stream *s* . Characters and strings are printed unquoted, and integers are printed in decimal.

(describe-object *o*:object *s* :stream**)** **Abstract Method**

`describe-object` prints information about object *o* onto stream *s*. `describe-object` prints more information than write, including the object's class and, for complex objects, contents.

(write-nnr *node* :integer *s*:stream **)** **Function**

`write-nnr` writes "`NNR`" onto stream *s* followed by the *x*, *y*, and *z* coordinates of the node number *node* .

(write-id *o*:object *s*:stream**)** **Macro**

`write-id` treats *o* as a word and writes its low-level representation onto stream *s*. The word is written as *tag*:*data*, where *tag* and *data* are integers. See Section 3.16.

(write-brackets *o*:object *body* **)** **Macro**

`write-brackets` prints "#<", executes *body*, and then prints a ">". The return value is unspecified. This macro is used to format complex objects.

Terminal Stream Operations

The following operations are provided as shorthands for working with `terminal-stream`.

(terpri) **Macro**

This function is the `terminal-stream` equivalent of the general stream method `terpri` above.

(print *object****)** **Macro**

`print` writes every *object* onto `terminal-stream` and returns the value of the last one, or `nil` if none were given.

(print1 *o*:object**)** :object **Function**

`print1` writes a newline, object *o*, and a space onto `terminal-stream` and returns *o*.

(vdescribe *o*:object**)** :object **Function**

`vdescribe` writes a description (from `describe-object`) of object *o* on a separate line onto `terminal-stream` and returns *o*.

3.15. Macros

Concurrent Smalltalk provides a macro facility which can be used to extend the language. A macro consists of a pattern, an optional guard, and a replacement. The pattern can contain variables or literals (a literal is an identifier). If it matches with an expression and the guard is satisfied, that expression is replaced by the replacement, which can be either another pattern or a Common Lisp function.

(defmacro *pattern* [*guard*] *replacement* **)** **Top-Level Macro**
pattern ::= *literal* | **?***name* | **!***name* | **(***pattern** [*pattern* **.** *pattern*]**)** | **@***pattern*
replacement ::= *r-pattern* | *lisp-statement*
r-pattern ::= *literal* | **?***name* | **!***name* | **(***r-pattern** [*r-pattern* **.** *r-pattern*] **)**
guard ::= **&guard** *lisp-statement*
lisp-statement ::= **#L** *lisp*

The macro pattern is a nested list of literals and macro variables. Variables are preceded by question marks (`?`) or exclamation points (`!`). Question-mark variables can match identifiers, numbers, and lists, while exclamation-point variables can only match identifiers. The dotted notation at the end of a list indicates that the rest of the list should match the pattern after the dot. When a pattern is matched to a candidate statement, all instances of the same variable have to match identical forms. The pattern can be as simple as `?x`, which will match any statement.

If an `@` symbol precedes a pattern, the form to which the pattern would match is macro-expanded before it is matched to the pattern. To avoid infinite loops, `@` should not be the first symbol in a macro pattern.

The guard, if present, is a Common Lisp statement that returns a boolean value. If the value returned is `true`, the macro replacement is substituted for the pattern; if not, the macro is not expanded. The values of the `?` and `!` variables are bound in a Common Lisp scope just outside the statement, so the Common Lisp statements can refer to the matched values of the variables just by referring to the correct variable names (including the leading `?` or `!`).

Replacement can be either another pattern or another block of Common Lisp statements. If *replacement* is a pattern, the values of the macro variables are substituted in it, and the replacement pattern replaces the original pattern in the code. If *replacement* is a Common Lisp statement, it is expected to return a list which replaces the original pattern in the code. As in the case of a guard, the Common Lisp statement has access to the matched values of the macro variables.

The macro replacement pattern can be another macro. Macros are expanded until the resulting form does not satisfy any of the existing macro patterns and guards. When several macros match a form, the form is expanded using the macro that was most recently defined.

3.16. Low-Level Operations

These operations provide access to low-level MDP and Cosmos operations. They are intended mainly for system and library development; use them with caution. Most of these operations are emulated either partially or not at all by the interpreter in Optimist II.

Constants

`_max-call-msg-size`:integer **`Constant`**

`_max-call-msg-size` is the maximum number of words allowed in a function or method call message. This does not include the initial destination node number word. Note that Cosmos's overhead for such messages is two words for the header, one word for each parameter, and two words for each continuation.

User programs can rely on this number being at least 72.

`(asm-word` *cosmos-constant* [*optimist-constant*]**`)`** :object **`Primitive`**
`(asm-int` *cosmos-constant* [*optimist-constant*]**`)`** :integer **`Primitive`**
cosmos-constant ::= *string*
optimist-constant ::= *expression*

`asm-word` returns the value of a link-time assembler expression, which must be given as a string literal. The expression can be an arbitrary tagged word expression that is valid for an MDPSim `DC` directive [4].

The optional *optimist-constant* is a top-level Concurrent Smalltalk expression that specifies a value that Optimist *may* [1] substitute for the constant when interpreting this primitive. It is useful for providing dummy values to let Optimist interpret programs which perform calculations on link-time constants. *Optimist-constant* must be a valid top-level expression—it is evaluated outside the scope of this expression, and should not perform any side effects or rely on partial results from the currently evaluated top-level expression. If *optimist-constant* is not specified, Optimist will leave the `asm-word` or `asm-int` value unevaluated when interpreting code.

Examples:

```
(+ (asm-word "4" 7) 1) ;Returns 8 in interpreter, 5 on J-Machine!
(asm-word "unchecked|Reply<<offsetN")
(asm-word "{foo}")
(asm-word "IP:abs|fault|unchecked|BlockMove<<offsetN")
```

Caution: The word generated by *cosmos-constant* must not have one of the `INST`*n* tags. Also, in order to keep the object file cube-size-independent, the expression should not rely, directly or indirectly, on the following MDPSim constants: `MEMSIZE`, `XNODES`, `YNODES`, `ZNODES`, `NNODES`, `LGXNODES`, `LGYNODES`, `LGZNODES`, `LGNNODES`.

`asm-int` is the same as `asm-word` except that it tells the Optimist compiler that the expression evaluates to an integer, which improves some optimizations.

[1] Optimist might not perform this substitution if the value is not passed to any functions.

Integers

(_lsh *n1* :integer *n2* :integer**)** :integer **Primitive**

`_lsh` returns *n1* shifted to the left by *n2* bits; if *n2* is negative, the shift is to the right. All bits shifted in are zero; if any one bits are shifted off the most significant end, an overflow occurs.

(_rot *n1* :integer *n2* :integer**)** :integer **Primitive**

`_rot` returns *n1* rotated to the left by *n2* mod 32 bits; *n2* can be negative. The bits rotated out are rotated back in from the other end.

(_ffb *n*:integer**)** :integer **Primitive**

`_ffb` returns the MDP `FFB` instruction applied to *n*. The result is the maximum amount by which *n* can be shifted to the left without causing an overflow of a 32-bit signed word; if *n* is 0 or -1, the result is 31. `(_ffb` *n* `)` is equal to `(- 31 (integer-length` *n*`))` but faster.

Tags

The main tag operations are listed below; all of these compile to fast, inline MDP code.

_tTAG0:integer **Constant**
_tINT:integer **Constant**
_tBOOL:integer **Constant**
_tADDR:integer **Constant**
_tIP:integer **Constant**
_tMSG:integer **Constant**
_tOBJ:integer **Constant**
_tCFUT:integer **Constant**
_tFUT:integer **Constant**
_tID:integer **Constant**
_tDID:integer **Constant**
_tTAGA:integer **Constant**
_tFLOAT:integer **Constant**
_tINST0:integer **Constant**
_tINST1:integer **Constant**
_tINST2:integer **Constant**
_tINST3:integer **Constant**

These constants provide names for the MDP tags. They have integer values between 0 and 15. `_tMSG` is the same as `_tOBJ`.

(_rtag *o*:object**)** :integer **Primitive**

`_rtag` returns the MDP tag field of the word *o* as an integer between 0 and 15.

(_value *o*:object**)** :integer **Macro**

`_value` returns the MDP value field of the word *o* as a 32-bit signed integer.

(_check *o*:object *tag* :integer**)** :boolean **Primitive**

`_check` returns `true` if word *o*'s tag matches the given tag number and `false` otherwise.

(_wtag *o*:object *tag* :integer**)** :object **Primitive**

`_wtag` changes the word *o*'s tag to *tag*, which is treated modulo 16. *o* does not have to be an integer.

_stSYM:integer **Constant**
_stCLASS:integer **Constant**
_stSEL:integer **Constant**
_stCHAR:integer **Constant**

These constants provide names for the MDP subtags of `TAG0`. They have integer values between 0 and 15.

(_rsubtag *o*:object**)** :integer **Primitive**

`_rsubtag` returns the MDP subtag field (bits 28 through 31, inclusive) of the word *o* as an integer between 0 and 15. The subtag field distinguishes several kinds of words which share the tag `TAG0`.

(_checksubtag *o*:object *subtag* :integer**)** :boolean **Function**

`_checksubtag` returns `true` if word *o*'s tag is `TAG0` and its subtag matches the given subtag number and `false` otherwise.

Physical Segments

physical-segment:primitive-class **Class**

Physical segments are absolute address/limit pairs that denote a range of memory words on a node. All physical segment operations refer to the local node only. Physical segment descriptors can be manually constructed using the `physical-segment` macro, or the segments can be allocated using `physical-new` or `disposable-physical-new`.

(physical-segment *address* :integer *limit*:integer **)** :physical-segment **Macro**
(physical-segment *cosmos-constant cosmos-constant* **)** :physical-segment **Macro**
cosmos-constant ::= *integer* | *string*

`physical-segment` constructs a nonrelocatable physical segment descriptor with the given MDP address and limit fields. The address is an absolute MDP memory address. These fields can be either integer expressions or Cosmos assembly language constants [4], which are either integer or string literals.

For example,

`(physical-segment 7 (+ x y))` and
`(physical-segment "NNRNodes" 6)`

are both valid (assuming the `NNRNodes` label is defined in MDPSim when the program is linked), but

```
(physical-segment "NNRNodes" (+ x y))
```

is not valid. One can also use MDPSim expressions within quotes, as in:

```
(physical-segment "NNRNodes" "NNRNodesSize+1")
```

Allocation

(physical-new *size* :integer**)** :physical-segment **Primitive**
(disposable-physical-new *size* :integer**)** :physical-segment **Primitive**

`physical-new` allocates a new segment with *size* words memory on the local node and returns a physical segment descriptor for it. The segment's words have offsets 0 through *size* -1, inclusive, and can all be used for user data. The first word is initialized to an integer holding the length of the segment but can be overwritten if desired. The segment descriptor will have a tight, nonzero limit field unless the size exceeds 1023.

`disposable-physical-new` is similar to `physical-new`, except that the segments it allocates can be disposed later. Also, the size of segments allocated with `disposable-`

`physical-new` can be rounded up by the operating system, and the first word of segments that will later be disposed must not be modified.

(physical-dispose *s*:physical-segment **)** :null **Primitive**

`physical-dispose` deallocates a segment allocated with `disposable-physical-new`.

Since physical segments are allocated from the top of memory and cannot be relocated and the operating system itself allocates physical segments at times, careless use of `disposable-physical-new` and `physical-dispose` is likely to lead to memory fragmentation.

References

(get-physical *s*:physical-segment *slot*:integer **)** :object **Primitive**

`get-physical` returns the word at the given offset in the physical segment *s*. A limit fault occurs if *slot* is not an integer between 0 and the size of *s* minus one (unless the object is larger than 1023 words).

(set (get-physical *s*:physical-segment *slot*:integer **)** *value*:object **)** :null **Primitive**

`put-physical` sets the contents of the word at the given offset in the physical segment *s* to *value* . A limit fault occurs if *slot* is not an integer between 0 and the size of *o* minus one (unless the object is larger than 1023 words, in which case a bad offset can corrupt memory).

Objects

_object-header:primitive-class **Class**

This class represents MDP words that are used as object headers. An object header includes information about the object's class as well as a number of Cosmos -specific flags pertaining to the object. Some of these flags are used for internal bookkeeping and may be changed by Cosmos at any time.

Allocation

(new-size *h*:object-header *size* :integer**)** :object **Macro**
(local-new-size *h*:object-header *size* :integer**)** :object **Function**
(new-on-size *node* :integer *h*:object-header *size* :integer**)** :object **Macro**
(new-size @*node* *h*:object-header *size* :integer**)** :object **Macro**

`new-size` creates and returns a new relocatable object with the given header and physical size (including headers) on a random node. Only the object's header is initialized—the data is not. The header and size for a standard class can be obtained with the `_instance-header` and `_instance-size` functions described later in this section.

`local-new-size` is the same as `new-size` except that the object is created on the local node. `new-on-size` creates the object on the specified node. The node number must be valid.

The last form, `(new-size @`*node h size*`)`, is just an alternate syntax for `new-on-size`.

See `new`, `local-new`, and `new-on` in Section 3.8 for higher-level operations to allocate objects.

(new-size *h*:object-header *size* :integer *n-constituents* :integer**)** :distobj **Macro**

`new-size` creates and returns a new distributed object with each constituent having the given header and physical size (including headers). Only the constituents' headers are initialized—their data is not. The header and size for a distributed class can be obtained with the `_instance-header` and `_instance-size` functions described later in this section.

References

`_object-header-slot-num`:integer **Constant**
`_object-size-slot-num`:integer **Constant**
`_instance-id-slot-num`:integer **Constant**
`_first-instance-slot-num`:integer **Constant**

These constants provide offsets of the Cosmos object header, size, ID slots and first data slot for use with `_ivar` and `put-_ivar`.

The following functions can be used to directly access objects' contents. They compile into efficient, inline code. When working with several objects, try to group references to a particular object together to avoid thrashing of the MDP's address registers.

Warning : The object *o* below should have metaclass `standard-class` or `distributed-class` or be a member of the `function`, `closure`, `selector`, one of the `class` classes, or one of the `array` classes. The low-level primitives below may fail in bizarre ways if applied to other primitive objects.

`(_ivar` *o*:object *slot* :integer**`)`** :object **Primitive**

`_ivar` returns the contents of the given slot of object *o*. A limit fault occurs if *slot* is not an integer between 0 and the size of *o* minus one (unless the object is larger than 1023 words).

`(set (_ivar` *o*:object *slot* :integer**`)`** *value* :object**`)`** :null **Primitive**

`put-_ivar` sets the contents of the given slot of object *o* to *value*. A limit fault occurs if *slot* is not an integer between 0 and the size of *o* minus one (unless the object is larger than 1023 words, in which case a bad offset can corrupt memory).

The results of using this function to modify immutable objects are unpredictable; Cosmos assumes that it can cache copies of immutable objects at will.

`(_object-header` *o*:object**`)`** :object-header **Macro**

`_object-header` returns the contents of the header slot of object *o*, which is always a member of the `_object-header` class.

`(_object-size` *o*:object**`)`** :integer **Macro**

`_object-size` returns the physical size, including headers, of object *o*.

Class Inquiries

`(_instance-header` *c*:class**`)`** :object-header **Macro**

`_instance-header` returns instance header of objects of class *c*. If the class's objects are one-word primitive objects and don't have instance headers, `_instance-header` returns `nil`. For classes such as `number` which allow creation of instance objects only indirectly through subclasses, `_instance-header` can return either `nil` or an object header.

`(_instance-size` *c*:class**`)`** :object-header **Macro**

`_instance-size` returns the physical size (including headers) of objects of class *c*. If the class's objects are one-word primitive objects and don't have instance headers, or if they have variable size as in arrays and functions, `_instance-size` returns `nil`.

Arrays

_simple-array-first-value-slot-num:integer **Constant**
_packed-array-first-value-slot-num:integer **Constant**
_packed-array-size-slot-num:integer **Constant**

These constants provide offsets of the first array data slots in simple and packed arrays and the number of elements slot in packed arrays for use with `_ivar` and `put-_ivar`.

_boolean-per-word:integer **Constant**
_char-per-word:integer **Constant**

These constants provide the number of `booleans` and `characters` packed into a single word of packed arrays.

3.17. Environment

Errors

(error (*msg* :object)* **)** **Function**

`error` signals a run-time error. The arguments, if any, should contain descriptive information about the error. The interpretation of the arguments' values is implementation-dependent.

(halt) **Primitive**

`halt` halts execution of the current program due to a run-time error. Debugging information about the function or method in which the halt took place may be printed.

Chapter 4. Using Optimist II

This chapter describes the procedure for using the Optimist II compiler to compile Concurrent Smalltalk programs. In addition, a few helpful Optimist II debugging commands are described.

Starting the Compiler

To start the compiler interactively, load the image containing the compiler and the Common Lisp envi ronment; on the Suns you can do this by running `optimist` with no parameters from the UNIX shell. If such an image is not available, load Common Lisp, CLOS, and the `Optimist.Lisp` file. If running on the Suns, make sure you are using Lucid Common Lisp 4.1.0 or later! Execute the `(optimist:compile-optimist)` command to compile and load the compiler, or, if it was already compiled, use `(optimist:load-optimist)` to load the compiler.

The compiler provides only one useful external Lisp function. It is `(interactive-cst)`. Typing `(interactive-cst)` will enter an interactive Concurrent Smalltalk listener loop. It is also possible to build a stand-alone executable of Optimist on the Suns using `(optimist:build-optimist nil "`*file-name*`")` or, if one is already loaded into memory, `(optimist:save-optimist "`*file-name*`")`.

Batch Mode Compilation

You can specify the arguments listed below on Optimist's UNIX shell command line. If you don't specify any arguments, Optimist will start up in interactive mode, running `(interactive-cst)`.

file-name — Specify the input file to be compiled. At least one input file must be specified. Multiple input files are allowed and are read sequentially.

`-o` *file-name* — Specify the assembly language output file. If no file is specified, the output is sent to the standard output stream. At most one `-o` can be used.

`-r` *function* — Specify a root function for compiling (see the `compile` command in the next section). *function* must be a function name. At least one root must be specified. Multiple roots can be specified by preceding each one with `-r`.

`-p` — Turn on progress messages as generated by the `progress` option later in this chapter. Progress messages default to off for batch mode compilation.

Top-Level Commands

Utility Commands

(begin *body* **)** **Top-Level Primitive**

Due to constraints in the compiler, a select few forms such as `include` and `defclass` (but not all of the top-level primitives; most of the primitives listed as **top level** really only require that they not be included in any function or method) must be present at the top level. However, sometimes it is desirable to emit sequences of those directives as results of macros; to allow this, a special form of `begin` was provided. If `begin` appears at the top level, every form inside it is also evaluated at the top level.

(set *name expression* **)** **Top-Level Macro**

`set` normally sets the variable *name* to the value of *expression*. However, if it is placed at the top level, it is also allowed to create a new global variable *name* if one does not exist already. Thus, at the top level, `set` acts as either `set` or `defglobal`, depending on whether the global variable *name* already exists.

(cd "*directory-name***")** **Top-Level Primitive**

`cd` sets the current default directory for reading and writing files.

(include "*file-name* **")** **Top-Level Primitive**
(include) **Top-Level Primitive**

`include` reads the definitions in the file named *file-name*, as if that file were included in place of the `include` primitive. `include`, when passed no file argument, will let the user interactively choose a text file and then include it. The latter feature is only available on the Macintosh version of Optimist II.

The default directory for *file-name* is the current directory or, if the `include` is embedded in a file currently being read in, the directory containing that file.

(quit) **Top-Level Primitive**

`quit` quits the Lisp environment of Optimist II. On the Suns you will be returned to the UNIX shell.

Viewing Objects

While the listener loop is active, any Concurrent Smalltalk command will be immediately evaluated, and the results displayed in the listener window. The resulting object may be displayed in a somewhat strange syntax; for example, integers may be displayed as `#<Integer 5>`, and booleans as `#<True>` or `#<False>`. The following commands may be used to show the internal structure of objects:

(show *o*:object**)** :object **Top-Level Primitive**

`show` shows as a side effect the Optimist II internal representation of an object. If the object is a function, its hcodes are shown; if the object is a complex object, some of its structure may be shown. The output is controlled by the CLOS `show` generic function. The value of the `show` directive is the object itself, so the object is usually printed normally after it is shown.

Please note that the hcodes shown for a function are only an approximation of the actual hcode data structure used internally to represent the function. Some of the more esoteric fields are not shown, and sometimes a function may have two variables with the same name, which leads to confusing output. Variable names were included for human readability only; Optimist II does not use them internally. It is able to keep the variables distinct regardless of their names. Also, since a digraph is a nonlinear structure, pseudo-hcodes such as jump labels in conditionals and `jump`, `label`, and `break` hcodes are inserted into the output to make it readable.

(describe *o*:object**)** :object **Top-Level Primitive**

`describe` describes as a side effect the Optimist II internal representation of an object. It is just like `show` except that the information displayed is longer and more detailed.

(show-hcode *f*:function [**#L***lisp-function-name*]**)** :function **Top-Level Primitive**

`show-hcode` calls Optimist's non-MDP-specific optimizations to optimize the function and shows the resulting hcode. `show-hcode` may invoke global optimizations and try to in line the functions called by *f*, so this directive may take some time to execute. When the `progress` option is true (the default), progress information is displayed in the listener window while this directive is executing. Detailed progress can be obtained by setting the `detailed-progress` option. `show-hcode` performs no side effects on the Concurrent Smalltalk environment, and it does not do a treewalk of the Concurrent Smalltalk program. `show-hcode` returns *f* as its result.

If *lisp-function-name* is provided, instead of showing the optimized hcode, `show-hcode` calls the Lisp function *lisp-function-name* with the optimized hcode as an argument. `describe-dinodes` is a useful Lisp function that will describe the compiled hcode in a little more detail.

`show-hcode` will not optimize a selector. If viewing optimized method code is desired, the method must be extracted explicitly using the Concurrent Smalltalk `method` primitive.

(show-mdp-hcode *f*:function [**#L***lisp-function-name*]**)** :function **Top-Level Primitive**

`show-mdp-hcode` is just like `show-hcode` except that it also performs the MDP-specific hcode optimizations.

(show-asm *f*:function [**#L***lisp-function-name*]**)** :function **Top-Level Primitive**

`show-asm` compiles the function *f* all the way to assembly code and prints the resulting MDPSim-compatible text. If *lisp-function-name* is supplied, it is assumed to be a lisp function and called with the assembly language module as its only argument.

Compiling Programs

(compile *object* ["*output-file-name*"]**)** :object **Top-Level Primitive**
(compile (*object****)** ["*output-file-name*"]**)** :object **Top-Level Primitive**

`compile` compiles and treewalks the Concurrent Smalltalk data structures starting with *object* as a root. Normally *object* is a function, in which case it is compiled to assembly language along with any other functions that it might need. If *output-file-name* is specified, the MDPSim file is written to a new file named *output-file-name*; otherwise, the output is sent to the listener. When the `progress` option is true (the default for interactive compilation), progress information is displayed in the listener window while this directive is executing. Detailed progress can be obtained by setting the `detailed-progress` option.

The second form of `compile` allows the specification of many roots. Note that if *object* is an expression, the second form must be used.

Options

(pragma …**)** — **Top-Level Primitive**

`pragma` is a general compiler declaration and can contain any implementation-dependent information.

(declare *option* *value***)** — **Top-Level Primitive**

`declare` sets the compiler option named *option* to the value specified. *Value* must be a legal value for the option; most compiler options are booleans, and for these *value* must evaluate to either `true` or `false`. *Value* must be a constant expression.

(option *option* **)** — **Primitive**

`option` returns the compile-time value of the specified compiler option.

Concurrent Smalltalk options can be set using the `declare` primitive and examined using the `option` primitive. The options currently provided by Optimist II are listed in Table 4-1.

Table 4-1. Options

Option	Default	Action
`n-nodes`	`4`	Define the number of nodes of a simulated J-Machine. This option only affects Optimist's internal interpreter; the compiled code is generic and will work on a J-Machine of any size, as long as the dimensions are powers of two.
`precise`	`false`	Inhibit optimizations that would affect the semantics of writing to constants in a few esoteric cases. This option should be set to `false` for normal use.
`delete-dead-defs`	`true`	Remove assignments to variables that will not be used again.
`delete-moves`	`true`	Try to remove unnecessary `move` intermediate compiler language statements.
`delete-touches`	`true`	Try to remove unnecessary `touch` intermediate compiler language statements.
`vflow-optimizations`	`true`	Calculate dataflow information and use it to perform a variety of optimizations such as changing $x \leftarrow y=0$, branch if x false sequences to `BNE` instructions.
`vflow-ivar-optimizations`	`false`	Use dataflow information to convert $i \leftarrow a$; …; $f(i)$ to $f(a)$ where a is an instance or global variable, i is a local, and … doesn't contain any function calls. This is a dangerous optimization and may introduce semantic errors or expand code.
`fold-constants`	`true`	Fold constants. For example, replace 1+2 by 3. Also remove conditional branches when it can be determined that the condition is always true or always false.
`fold-global-constants`	`true`	Fold constants globally. For example, replace a call through a selector with a call of the method when the method can be determined using type analysis. This option is relevant only when fold-constants is true.

`forward-tails` `true`
: Enable the altering of `application` hcodes immediately followed by returns into tail-forwarded `applications` which allow the process to be deallocated and the answer directly forwarded to the caller. This is the equivalent of tail recursion.

`merge-code` `true`
: Merge common pieces of code wherever possible.

`inline` `true`
: Inline small functions.

`inline-size-cutoff` `15`
: Set the size cutoff for automatically deciding whether to inline a function. Increasing this number causes larger functions to be inlined.

`local-inline-size-cutoff` `60`
: Set the size cutoff for automatically deciding whether to inline a function which is known to execute on the same node as it was called from. Increasing this number causes larger functions to be inlined.

`max-function-size` `450`
: Set the maximum size for deciding whether a function is too large already to allow any inlining to take place inside the function. Increasing this number will cause the compiler to be more aggressive about performing inlining in larger functions.

`apply-default-local-node` `false`
: If `true`, function calls default to being made on the local node rather than the node of the first argument.

`optimize-built-ins` `true`
: Perform local built-in optimizations such as changing multiplications to shifts.

`optimize-float-library-calls` `true`
: Replace instructions such as `ADD` and `MUL` that fault on `floats` with direct calls to the floating point library when the arguments are known to have type `float`.

`compact-vars` `true`
: Compact variables in the context to use as few slots as possible.

`reg-variables` `true`
: Assign variables to registers whenever possible.

`lru-register-allocation` `true`
: Use the least-recently-used algorithm to allocate temporary registers during code generation.

`frame-regs` `true`
: Keep track of variables in the registers during code generation and use values from the registers instead of from memory whenever possible.

`frame-migrate` `true`
: Keep track of whether it is possible for the instance object to have migrated away. Don't touch it if it could not have migrated away.

`lazy-ivar-access` `true`
: Don't `XLATE` the instance object if there are no references to it.

`lazy-contexts` `true`
: Don't allocate a context unless it is actually used.

`fast-contexts` `true`
Use fast contexts whenever possible.

`optimize-send-self` `true`
Send message to the current node if the receiver is `self` and it is not atomic.

`fast-apply` `true`
Use `ApplyFunction` and `ApplySelector` instead of `Apply` whenever possible.

`compact-sends` `true`
Try to combine `SEND`s and `SENDE`s into `SEND2`s and `SEND2E`s.

`compact-DCs` `true`
Try to align `DC`s on word boundaries whenever possible.

`optimize-br-to-suspend` `true`
Replace unconditional branches to `SUSPEND`s by `SUSPEND` instructions.

`delete-locals` `true`
Delete local variables in an intermediate stage of the compilation. This makes no difference in the final output, but makes the hcode look prettier and may speed up code generation.

`warn-free-references` `false`
Emit a warning every time a free reference is found in a method or function.

`progress` `true`
Print progress reports. Progress messages default to off for batch mode compilation.

`detailed-progress` `false`
Print very long progress reports.

`monitor-compiler` `false`
Monitor time spent in different parts of compiler.

`print-pc` `true`
Print program counter values as comments in output.

`assembly-comments` `false`
Insert hcode and other comments in assembly output.

`permanent-definitions` `false`
Use `defconstant` instead of `defparameter` when compiling function and method definitions. When this option is set, a warning is emitted every time a free reference is found in a method or function regardless of the setting of warn-free-references.

`lisp-break` `true`
Enter a Common Lisp break loop upon a Concurrent Smalltalk warning or error.

Bibliography

[1] Waldemar Horwat. "AIDA: Data Structure Optimization on a MIMD Parallel Computer." *Proceedings of the 1992 MIT Student Workshop on VLSI and Parallel Systems*, Charles Leiserson, ed., 1992, pp. 17-1–17-2.

[2] Waldemar Horwat. *A Concurrent Smalltalk Compiler for the Message-Driven Processor*. MIT Artificial Intelligence Laboratory Technical Report 1080, May 1988.

[3] Waldemar Horwat. *Concurrent Smalltalk on the Message-Driven Processor*. MIT Artificial Intelligence Laboratory Technical Report 1321, September 1991.

[4] Waldemar Horwat. *Message-Driven Processor Simulator, Version 7.2c*. MIT Concurrent VLSI Architecture Memo 53, February 1993.

[5] Waldemar Horwat. *Revised COSMOS Manual*. MIT Concurrent VLSI Architecture Memo 52, April 1993.

[6] Guy L. Steele, Jr. *Common Lisp: The Language*, Second Edition. Digital Press, Digital Equipment Corporation, 1990.

Index

" 15
^ 13
! 13, 14, 54
#' 14, 15
#: 14, 15
#L 54
#\ 15
% 13
& 13, 14
&cas-er 20
&cwriter 20
&guard 54
&immutable 20
&inline 22
&name 21
&no-leak 21
&non-strict 22, 25
&not-inline 22
&predicate 20
&reader 20
&side-effect-free 22
&value 21, 22
&writer 20
' 14
* 44
+ 44
- 44
/ 44
// 44
: 19, 21, 27, 28
:: 21
; 4, 16
< 42
<= 42
<> 42, 45
= 42, 45
> 42
>= 42
? 13, 14, 29, 54
@ 14, 54
abs 44
abstract class 12
abstract method 12
acquire 50
ADD 66
add-method 23
and 42, 43
short-circuit 43
Apple Common Lisp 1
application 26, 66
apply-default-local-node 27, 66
argument 3, 21
evaluation order 26
passing convention 21
array 1, 33, 47, 59
boolean, see boolean-array
character, see string
general, see general-array
integer, see integer-array
array? 37
ASCII 45
ash 45
asm-int 55
asm-word 55
assembly-comments 67
become 11
begin 6, 17, 25, 29, 63
big-computation 7
block 13, 22, 31
BNF 12
body 25
boolean 33
boolean-array 33, 47
boolean-array? 37
boolean-per-word, see _boolean-per-word
boolean? 37
buffered-output-stream 33, 51, 52
bug fixes 1
build-optimist 62
busy? 50
cap 29
cap- 20, 29
car 4, 46
cas 28, 29
cas-er method 20
cd 63
cdr 4, 46
cfuture 2, 8, 25, 26
semantics 25
changes 1-2
char-code 45
char-per-word, see _char-per-word
char< 42, 45
char<= 42, 45
char> 42, 45
char>= 42, 45
character 15, 33, 45
character? 37
check, see _check
checksubtag, see _checksubtag
class 4, 15, 19, 33, 59
abstract 12
assertion 27
built-in 19, 33
hierarchy 34
immutable 20
inheritance 19
inline 2
inquiry 37
metaclass 34
predicate 4, 20
class-kind? 37
class-name 38
class-of 38
class? 37
clet 6, 7, 13, 27
multiple value 28
clet* 1, 27
CLOS 62
closure 22, 33, 59
closure? 37
co 40, 41
code-char 45
collection 33
collection? 37
comment 16
comments 4
Common Lisp 1, 10, 11, 13, 21, 46, 54, 67
compact-DCs 67
compact-sends 67
compact-vars 66
compare-and-set, see cas
compile 62, 64
compile-optimist 62
compiler option 65
complex numbers 13
complex-number 12
concurrency 7
concurrently 7, 29
cond 1, 30
conditional 30
cons 4, 46
constant 17
expression 17
predefined 15
constituent 40
number 40
context future, see cfuture
context, see _context
continuation 22, 31-32
continuation, see also _continuation
copy 36
Cosmos 1, 51, 57, 58

cput 29
cput- 20, 29
cset 1, 6, 8, 28, 29
  multiple value 28
cwriter method 20
DC 67
declare 65
deep-copy 36
deep-dispose 36
defclass 4, 13, 19
defconstant 13, 17
defglobal 13, 17, 63
define 13, 18
defmacro 10, 54
defmethod 13, 17, 24, 31
defparameter 17
defselector 13, 17, 23
defun 3, 13, 17, 23, 31
delete-dead-defs 65
delete-locals 67
delete-moves 65
delete-touches 65
describe 64
describe-object 52, 53
detailed-progress 64, 67
direction-node 36, 39
directory 63
disposable-physical-new 57
dispose 37
distarray 9
distobj 9, 23, 33, 40
  creation 40, 58
distobj? 37
distributed object, see distobj
distributed-class 33, 34, 59
distributed-class? 37
divmod 1, 44
down 36
down-direction 39
east 36
east-direction 39
eq 12, 42
error 61
evaluation order 26
exit 31
exponent 13
expression 25
  constant 17
false 15, 33
false? 37
fast-apply 67
fast-contexts 67
feq 12
ffb, see _ffb
ffib 3
fib 3, 6
file 62, 63
fill 48
fill-range 48
float 1, 33, 42, 43, 45, 66
float? 37
floor 45
fold-constants 65
fold-global-constants 65
for 1, 30
for-each 48
force-output 51, 52
formal 21
  no-leak 21
  value 21
format 12
forward-tails 66
frame-migrate 66
frame-regs 66
funct 33
funct? 37
function 3, 4, 5, 21, 33, 59
  calling 22, 26
  inline 22
  non-strict 22
  not-inline 22
  predicate 4, 20
  return value 22, 31
  side-effect-free 22
function? 37
future 2, 25
garbage collection 1
general-array 33, 47
general-array? 37
get 28, 29, 48
get- 5, 20, 29
get-buffer-length 51
get-car 46
get-cdr 46
get-group 41
get-physical 58
get-port-number 51
get-underlying-stream 51
global 15, 17
goals 11
group 13, 23, 40, 41
halt 61
here 36
home-node 27, 39
identifier 13
  undefined 15
if 17, 30
immutable class 20
include 63
index 41
inheritance 4, 19
  multiple 19
init 48, 50
inline 22, 66
  class 2
inline-size-cutoff 66
input 2, 51
input file 62
instance
  object 4, 36, 58
  variable 4, 20, see also _ivar
instance-header, see _instance-header
instance-size, see _instance-size
integer 33, 42, 43
integer-array 33, 47
integer-array? 37
integer-length 45
integer-to-fp 45
integer? 37
interactive-cst 62
ivar, see _ivar
J-Machine 11
lambda 13, 22, 23, 31
large-integer 2
lazy-contexts 66
lazy-ivar-access 66
length 46
let 6, 7, 13, 27
  multiple value 28
let* 1, 28
lfib 6
LGNNODES 55
LGXNODES 55
LGYNODES 55
LGZNODES 55
lisp-break 67
list 15, 46
load-optimist 62
local variable 6
local-inline-size-cutoff 66
local-new 36, 46, 58
local-new-size 58
local-node 27, 36, 38
lock 8, 33, 50
logand 43
logical-limit 40
lognot 43
logor 43
logxor 43
loop 13, 30, 31
low-level operation 55
lru-register-allocation 66
lsh, see _lsh
Lucid Common Lisp 1, 62
Macintosh 1
macro 10, 12, 13, 54
  guard 54
magnitude 2
make-nnr 38
make-safe-nnr, see _make-safe-nnr

map 48
max 44
max-call-msg-size, see _max-call-msg-size
max-function-size 66
max-x-node 38
max-y-node 38
max-z-node 38
MDPSim 1, 55, 57, 64
MEMSIZE 55
merge-code 66
metaclass 34, 59
method 3, 4, 21, 23, 35, 64
    abstract 12
    built-in 34
    calling 22, 26
    cas-er 20
    cwriter 20
    inline 22
    non-strict 22
    not-inline 22
    overriding 5
    primitive 35
    reader 5, 20
    return value 22, 31
    side-effect-free 22
    writer 5, 20
method-lambda 13, 22, 23, 31
min 44
minus? 44
mod 44
monitor-compiler 67
move 65
MUL 66
multiple inheritance 19
multiple value 22, 28, 31
mv-clet 13, 28
mv-cset 28
mv-let 13, 28
mv-set 28
n-nodes 65
name 13
    undefined 15
name space 13
nconcurrently 7, 29
neg 44
neighbor 36
neighbor-node 36, 39
neq 42
new 36, 40, 46, 58
new-boolean-array 47
new-boolean-array-on 47
new-buffered-output-stream 51
new-general-array 47
new-general-array-on 47
new-integer-array 47
new-integer-array-on 47
new-on 36, 58
new-on-size 58
new-queueing-lock 50
new-simple-lock 50
new-simple-stream 51
new-size 58
new-string 47
new-string-on 47
nfor-each 48
nil 15, 33, 46
NNODES 55
NNR 27, 38
nnr-x 38
nnr-y 38
nnr-z 38
no-leak formal 21
node 27, 38, 65
node-mask, see _node-mask
non-strict 22, 25
north 36
north-direction 39
not 42, 43
not-inline 22
nparallel 2
null 15, 33
null? 37
number 13, 15, 33
    complex 13
number? 37
object 4, 5, 7, 33, 37
    class 37
    constituent 40
    creation 36, 58
    distributed, see distobj
    instance 4, 36, 58
object-header, see _object-header, see _object-header
object-size, see _object-size
Optimist 1, 62-67
optimize-br-to-suspend 67
optimize-built-ins 66
optimize-float-library-calls 43, 66
optimize-send-self 67
option 65
or 43
    short-circuit 43
output 2, 51
output file 62
overriding 5
packed-array 33, 47
packed-array? 37
pair 4, 7, 33, 46
parallel 2
parameter 3, 17, 21
    passing convention 21
permanent-definitions 67
physical-dispose 58
physical-limit 40, 41
physical-new 57
physical-segment 33, 57
physical-segment? 37
plus? 44
pragma 65
precise 65
predicate function 4, 20
primitive 12
primitive method 35
primitive selector 35
primitive-class 33, 34
primitive-class? 37
print 53
print-pc 67
print1 53
program 17
progress 62, 64, 67
put 29, 48
put- 5, 20, 29
put-car 46
put-cdr 46
put-physical 58
put-_ivar 59
queue overflow 1
queueing-lock 9, 33, 50
quit 63
quote 15, 46
random 36
random-node 36, 38
reader method 5, 20
real 33
real? 37
receiver 3, 23
reg-variables 66
release 50
repeat 30
reply 31, 32
resource 8
restricted selector 35
return 31, 32
return value 22, 31
    declaration 21
    multiple 22
return-value-expected? 32
root 62, 64
rot, see _rot
rsubtag, see _rsubtag
rtag, see _rtag
save-optimist 62
sc-and 43
sc-or 43
Scheme 11, 21
scope 13, 14
    static 22
selector 3, 33, 59
    primitive 2, 35
    restricted 2, 35

selector? 37
self 3, 13, 23, 40
SEND 67
SEND2 67
SEND2E 67
SENDE 67
set 1, 6, 28, 29, 63
  multiple value 28
shallow-copy 36
shallow-dispose 37
show 63
show-asm 64
show-hcode 64
show-mdp-hcode 64
side-effect-free 22
simple-array 9, 33, 47
simple-array? 37
simple-lock 9, 33, 50
simple-stream 33, 51
size 48
slot 59
small-integer 2
Smalltalk-80 11, 21
south 36
south-direction 39
sqrt 45
standard-class 33, 34, 59
standard-class? 37
statement 17, 25
  application 26
strcmp 49
stream 33, 51
strict 22, 25
string 15, 33, 47, 49
string-append 49
string< 49
string<= 49
string<> 49
string= 49
string> 49
string>= 49
string? 37
subclass 6
subclass? 38
subtag 57
subtype 6
superclass 6, 19
supertype 6
SUSPEND 67
symbol 15, 33
symbol? 37
tag 56
terminal-stream 51, 53
terpri 52, 53
tfib 7
time-sink 7
token 13
top-level form 17
touch 8, 25, 26, 65
true 15, 33
true? 37
type 4, 6
  assertion 27
  checking 6
  declaration 7
unbuffered-terminal-stream 51
undef 18
unless 1, 30
up 36
up-direction 39
value 21, 25
  formal 21
  multiple 22, 28, 31
  return 22, 31
    declaration 21
value, see also _value
variable
  instance 4, 20, see also _ivar
  local 6
  scope 22
vdescribe 53
vflow-ivar-optimizations 65
vflow-optimizations 65
warn-free-references 67
west 36
west-direction 39
when 1, 10, 30
while 30
with-locks 50
write 52
write-brackets 53
write-id 53
write-nnr 52
write-space 52
write-stream 52
write-stream-char 52
write-stream-string 52
writer method 5, 20
wtag, see _wtag
XLATE 66
XNODES 55
xor 43
YNODES 55
zero? 12, 43
ZNODES 55
[ ] 28
\ 15
_ 13, 21, 27
_boolean-per-word 60
_char-per-word 60
_check 56
_checksubtag 57
_context 33, 37
_continuation 33, 37
_ffb 56
_first-instance-slot-num 59
_instance-header 58, 59
_instance-id-slot-num 59
_instance-size 58, 59
_ivar 59
_loop 1, 31
_lsh 56
_make-safe-nnr 38
_max-call-msg-size 55
_node-mask 38
_object-header 33, 58, 59
_object-header-slot-num 59
_object-header? 37
_object-size 59
_object-size-slot-num 59
_packed-array-first-value-slot-num 60
_packed-array-size-slot-num 60
_rot 56
_rsubtag 57
_rtag 56
_simple-array-first-value-slot-num 60
_stCHAR 57
_stCLASS 57
_stSEL 57
_stSYM 57
_tADDR 56
_tBOOL 56
_tCFUT 56
_tDID 56
_tFLOAT 56
_tFUT 56
_tID 56
_tINST0 56
_tINST1 56
_tINST2 56
_tINST3 56
_tINT 56
_tIP 56
_tMSG 56
_tOBJ 56
_tTAG0 56
_tTAGA 56
_value 56
_wtag 56