Object-Oriented Programming in Common Lisp: A Tutorial

STD-N-XXXX
August 15, 1994
Marty Hall

The Johns Hopkins University
Applied Physics Laboratory
Johns Hopkins Rd.
Laurel, MD 20723-6099

Introduction

Object-oriented programming is becoming an increasingly important software technology in the Laboratory in general and in STD in particular. Object-oriented simulations are an especially important application of this technology, as illustrated by FLAIMS, ORBIS, and SMS, three recent STD systems.

The Common Lisp Object System (CLOS) is an advanced object-oriented language that is part of the Common Lisp standard, and is based upon Flavors, Common LOOPS, and, indirectly, Smalltalk. ORBIS, the Object-oriented Rule-Based Interactive Simulation, is an important simulation shell in STD, and has increasing levels of sponsor interest and funding. Although implemented in Lisp, it was created before the introduction of CLOS, and its object-oriented components are not consistent with today's standards. In order to stay competitive, future versions of ORBIS must be upgraded to make use of the latest object-oriented design and implementation strategies, and CLOS language features.

This tutorial was designed to introduce ORBIS developers to the major language features of CLOS, the types of applications these features support, and any particularly important usage or performance issues that are associated with those features. It would be appropriate for other software developers who are already familiar with the basics of Common Lisp and the core ideas of object-oriented languages. It is centered around a series of exercises that illustrate the various features of the language. Sample solutions for the exercises are included.

Outline

I) What Are Objects?
- A) Comparison to structures/records
- B) 3 facets
  - 1) Classes and inheritance
  - 2) Behavior associated with objects (methods/ overloading)
  - 3) Encapsulation and Data Protection
- C) Simple Example: Geometric Shapes

II) Defining Classes: The Basics
- A) defclass
- B) make-instance
- C) slot-value
- D) (setf slot-value)
- E) :initform
- F) :initarg
- G) Example: Geometric Shapes

III) Defining Methods: The Basics
- A) defmethod
- B) Single-argument methods
- C) Area
- D) Multimethods
- E) Concat

IV) Defining Classes: Some Details
- A) :reader
- B) :writer
- C) :accessor
- D) :allocation
- E) :type
- F) :documentation

V) Exercise: Warm-up

VI) Defining Methods: Some Details
- A) Imitating :reader and :writer
- B) Most specific method wins
- C) lambda list congruence
- D) Idea of fallback method to trap errors
- E) setf methods
- F) Imitating :accessor

VII) Common Blunders
- A) Omitting parens in arglist in defmethod
- B) Missing parens around slot definition list in defclass
- C) Forgetting empty slot definition list if you don’t define local slots in defclass
- D) Referring to class name instead of instance variable in defmethod
- E) Forgetting accessors are functions and thus could conflict with built-in function names
- F) Putting the new value last instead of first in the definition of a setf method
- G) Putting the new value last instead of first in a call to a :writer method
- H) Confusion about doc strings in defmethod
- I) Invalid :initargs accepted by make-instance
- J) Forgetting the class has to exist before any methods that specialize upon it
- K) Changing a method to apply to a more general class does not supersede previous method

VIII) Exercise: Setf Methods

IX) Simplified Interfaces
- A) Define-Class/Def-Class
- B) Example: Integrated Circuits/Gates

X) Defining Classes: More Details
- A) Class Options
  - 1) :documentation
  - 2) :default-initargs
  - 3) :metaclass
- B) initialize-instance
- C) print-object
- D) describe-object

XI) Exercise: print-object

XII) Defining Methods: More Details
- A) :before methods
- B) :after methods
- C) eql specializers

XIII) Intermediate Exercise: Blocks World

XIV) Miscellaneous Useful Topics
- A) Multiple Inheritance
- B) call-next-method and :around methods
- C) with-slots/with-accessors
- D) Lower-level type checking
  - 1) type-of
  - 2) typep
  - 3) subtypep
  - 4) typecase

XV) Exercise: Multiple Inheritance

XVI) Timing and Efficiency
- A) Slot access
- B) Function Call Overhead
- C) Instance Creation

XVII) Exercise: Timing and Efficiency

XVIII) defgeneric
- A) Giving overall arglist and documentation
- B) Defining multiple methods at once
- C) :argument-precedence-order
- D) :method-combination
- E) Example: showing class precedence list

XIX) Advanced Topics
- A) Unique names
  - 1) gentemp vs. closures vs. own constructor
  - 2) global list vs. "set" trick vs. hash tables with names as keys
- B) Dumping object to file
- C) MOP
- D) Copying instances

XX) Extended Exercise: Object-Oriented Simulation

Thumbnail CLOS Summary

■ defclass

Slot Options

- **`:initform`** - Default slot value. Evaluated at make-instance time.
- **`:reader`** - slot option to define a method to read a slot value
- **`:writer`** - makes method to set slot value
- **`:accessor`** - makes two methods, a reader and a setf writer
- **`:initarg`** - Defines a keyword to make-instance that sets slot value
- **`:allocation`** - determines if slot is shared (:class) or normal (omit or :instance)
- **`:type`** - Allows (but does *not* require) compiler to make type assumptions
- **`:documentation`** - Associates a doc string with a slot. No portable way of retrieving this info.

Class Options

- **`:documentation`** - Associates a doc string with a class. Use documentation on the class object to retrieve this info. You can use class-of, find-class, or the return value from defclass to get the class object.
- **`:default-initargs`** - Allows default values (ala :initform) of already existing :initargs without having to respecify slot.
- **`:metaclass`** - Allows customization of basic behavior of classes. Hackery for gurus only.

■ Miscellaneous Class Operations

- **`make-instance`** - Creates an actual object of a class.
- **`slot-value`** - Atomic function to read the value of a slot. You usually want to use accessors instead.
- **`(setf slot-value)`** - Atomic function to write the value of a slot. Normally use a writer or (setf Accessor) function, but this internal version would be called when writing your own custom writer.
- **`initialize-instance`** - Called by make-instance, and the best place to put :after methods to do bookkeeping on instances.
- **`print-object`** - Used whenever Lisp prints out an instance. You can write a custom version to override default.
- **`with-slots`** - a shorthand notation that lets you refer to slots as though they were variables, but really use slot-value each time the variable is referenced or set.
- **`with-accessors`** - a shorthand notation that lets you refer to accessors as though they were variables, but really calls the accessor function each time the variable is referenced or set.
- See **`Define-Class`** and **`Def-Class`** macros for simplified interface with automatic :initform, :initargs and :accessor.

■ defmethod

- Similar to defun but can replace a var in the arglist with a **`(Var Type)`** pair
- If no generic function exists when method defined, one is created
- Most built-in data types and all classes can be specialized upon
- Cannot specialize on **`&optional`** or **`&key`** args
- For primary methods, only the single most specific method is called
- For **`:before`** methods, all are executed in least-specific to most-specific order
- For **`:after`** methods, all are executed in most-specific to least-specific order
- **`:around`** methods are the only type that can prevent primary methods from firing
- **`setf forms`** are considered primary methods, and thus can have :before, :after, and :around methods associated with them
- Cannot add a method to a built-in standard (i.e. non-generic) function

■ defgeneric

- Defines generic function template
- Can define multiple methods in one form rather than via multiple uses of defmethod.
- **`:method-combination`** - Instead of having the single most specific method winning, you could combine applicable methods in various ways using this keyword.
- **`:argument-precedence-order`** - Defines which order arguments are examined in determining "most specific". Default is L to R.
- **`:documentation`** - Makes documentation for the generic function ala doc strings in defun
- Cannot change a standard function into a generic one.

DEFCLASS

■ Basic Syntax

```
(defclass Class-Name (Superclass*)
  (Slot-Definition*)
  Class-Option*)†
```

■ Examples

• Empty superclass list, empty slot definition list, no class options

```
(defclass Graphical-Object ()
  ())
```

• One superclass, two slot definition lists, no class options

```
(defclass Rectangle (Graphical-Object)
  ((Height :accessor Height :initarg :Height :initform 3)
   (Width  :accessor Width  :initarg :Width  :initform 5)))
```

• One superclass, no slots, one class option.

```
(defclass Square (Rectangle)
  ()
  (:documentation "The SQUARE class"))
```

■ make-instance and slot-value

Given a class definition, **`make-instance`** will make an actual object of that class. The **`slot-value`** function is the most primitive function for retrieving and setting values of slots.

```
(defclass Submarine ()
  ((Depth)
   (Speed)))

(setq Sub-1 (make-instance `Submarine))
(setf (slot-value Sub-1 'Depth) 100)
(slot-value Sub-1 'Depth) ⇒ 100
(slot-value Sub-1 'Speed) ⇒ [Error: unbound slot SPEED]
```

Unlike with the methods defined by **`:reader`**, **`:writer`**, and **`:accessor`**, you cannot put **`:before`** or **`:after`** methods on **`slot-value`** or **`(setf slot-value)`**.

There are also functions **`slot-exists-p`** and **`slot-boundp`** for checking if a given instance has a certain slot and if that slot has a value, respectively.

■ Slot Definitions

Slot definitions are lists composed of the slot name followed optionally by pairs of the following slot option keywords and associated value.

`:initform`	a default value for the slot (unbound otherwise)
`:reader`	the name of a function to lookup the slot
`:writer`	the name of a function to set the slot
`:accessor`	the name of a function to read, and (with **`setf`**) set the slot
`:initarg`	the name of a keyword for **`make-instance`**
`:allocation`	whether slot is shared or specific to each instance
`:type`	a Common Lisp type for use in compiler optimizations
`:documentation`	documentation for the very rarely-seen slot object

Values supplied by **`:initform`**, **`:allocation`**, and **`:documentation`** *override* any previously supplied value in a superclass. Values supplied via **`:reader`**, **`:writer`**, **`:accessor`** and **`:initarg`** give values that apply to the current class *in addition to* values supplied for superclasses. A **`:type`** restriction imposed on a slot is *intersected* with the restrictions on that slot imposed by superclasses. Similarly, you can only supply one **`:initform`**, **`:allocation`**, **`:documentation`**, or **`:type`** entry for a given slot, but you can supply multiple **`:reader`**, **`:writer`**, **`:accessor`** or **`:initarg`** entries.

† I'm using regular expression notation, where a "*" means "0 or more occurrences of". Thus (Superclass*) means "0 or more superclasses, inside parens".

■ :initform

Specifies a default initial value for the slot. This value is evaluated each time **make-instance** is called. Supplying an explicit initial value at **make-instance** time (see **:initarg**) will override this.

```
(defclass Submarine ()
  ((Depth :initform 100)
   (Speed :initform 5)))

(setq Sub-1 (make-instance 'Submarine))
(slot-value 'Sub-1 'Speed) ⇒ 5
(slot-value 'Sub-1 'Depth) ⇒ 100
```

■ :reader

Specifies the name of a function that will read the slot (in lieu of using **slot-value**). This value is NOT evaluated, and thus should not be quoted. Using **:accessor** obviates the need for **:reader** or **:writer**.

```
(defclass Submarine ()
  ((Depth :reader Depth :initform 100)
   (Speed :reader Speed :initform 5)))

(setq Sub-1 (make-instance 'Submarine))
(Depth Sub-1) ⇒ 100
(Speed Sub-1) ⇒ 5
```

■ :writer

Specifies the name of a function that will set the slot (in lieu of using **setf** on **slot-value**). Not evaluated. Using **:accessor** obviates the need for **:reader** or **:writer**.

```
(defclass Submarine ()
  ((Depth :reader Depth :writer Set-Depth :initform 100)
   (Speed :reader Speed :writer Set-Speed :initform 5)))

(setq Sub-1 (make-instance 'Submarine))
(Depth Sub-1) ⇒ 100
(Speed Sub-1) ⇒ 5
(Set-Depth 200 Sub-1)
(Set-Speed 4 Sub-1)
(Depth Sub-1) ⇒ 200
(Speed Sub-1) ⇒ 4
```

■ :accessor

Specifies the name of a function that will both read and (with **setf**) set the slot. If you have a **:accessor**, you don't need to specify either **:reader** or **:writer**. Not evaluated.

```
(defclass Submarine ()
  ((Depth :accessor Depth :initform 100)
   (Speed :accessor Speed :initform 5)))

(setq Sub-1 (make-instance 'Submarine))
(Depth Sub-1) ⇒ 100
(Speed Sub-1) ⇒ 5
(setf (Depth Sub-1) 200)
(setf (Speed Sub-1) 4)
(Depth Sub-1) ⇒ 200
(Speed Sub-1) ⇒ 4
```

■ :initarg

Specifies a keyword that can be passed to **make-instance** to give a slot an initial value. If this keyword is used at **make-instance** time, it overrides any default initial value that **:initform** provided. Unlike with **defstruct**, **defclass** does not make any of these keywords automatically available.

```
(defclass Submarine ()
  ((Depth :accessor Depth :initarg :Depth :initform 100)
   (Speed :accessor Speed :initarg :Speed :initform 5)))

(setq Sub-1 (make-instance 'Submarine :Depth 200))
(Depth Sub-1) ⇒ 200
(Speed Sub-1) ⇒ 5
```

■ :allocation

Specifies whether a slot's value is specific to each instance (**:instance**), or has a single value that is shared by all instances of that class (**:class**). The default is **:instance**.

```
(defclass Submarine ()
  ((Depth :accessor Depth :initform 100)
   (Speed :accessor Speed :initform 5)))

(defclass Los-Angeles (Submarine)
  ((Max-Speed :accessor Max-Speed :initform 10 :allocation :class)))

(setq SSN-1 (make-instance 'Los-Angeles))
(setq SSN-2 (make-instance 'Los-Angeles))
(Max-Speed SSN-1) ⇒ 10
(Max-Speed SSN-2) ⇒ 10
(setf (Max-Speed SSN-1) 15)
(Max-Speed SSN-1) ⇒ 15
(Max-Speed SSN-2) ⇒ 15
```

■ :type

Specifies a Common Lisp type for the value of the slot. This type information may be used by the compiler to provide optimizations, or might be used to do error checking. However, error checking is not required, and probably won't be done except when compiled with high **safety** settings. It is similar to a type declaration in C: putting a value of the wrong type into a slot may result in a relatively arbitrary error next time the slot value is used. Also recall that, in Common Lisp in general, type **integer** doesn't let the compiler do many optimizations, since integers can be of arbitrary size. For fixed-size integers, like **int**'s in C, use the **fixnum** type declaration.

The **:type** entry gets inherited by combining the type restrictions of the current class and all superclasses using **and**. This means that a class cannot relax a type restriction imposed by a superclass, only make it more stringent.

```
(defclass Submarine ()    ((Depth :type integer)))
(defclass SSN (Submarine) ((Depth :type fixnum)))
```

This means that a legal value of the **Depth** slot must satisfy **(typep** *value* **'(and integer fixnum))**.

■ :documentation

This option, rarely used, specifies documentation for the slot. The only way to retrieve this documentation is to call the **documentation** function on the slot object. However, the slot object can only be gotten by using a function in the CLOS MOP (Meta-Object Protocol). Since the MOP did not get accepted as part of the first ANSI standard for Lisp, this means that there is no way in standard Lisp to retrieve the slot documentation! Now, the simplest MOP functions are in fact in most implementations (including Lucid, Symbolics, Harlequin, and Franz), but since this is not guaranteed to be portable, slot documentation is generally not used.

■ Class Options

Class options are a list of one of the following keywords, along with an associated value.

:documentation	documentation for the class object
:default-initargs	default values for the previously specified **:initarg**'s
:metaclass	the superclass of the class object itself. Leave this for the CLOS wizards.

Each class option can be specified at most once for a given class.

■ :documentation

Specifies a documentation string for the class object. Passing the class object to the **documentation** function will retrieve it. The class object is returned from a call to **defclass**, or can be found by passing an instance to **class-of** or supplying a class name to **find-class**.

```
(setq Submarine-Class
  (defclass Submarine ()
    ((Depth :accessor Depth :initform 100)
     (Speed :accessor Speed :initform 5))
```

```
    (:documentation "The basic submarine class")) )

(setq Sub-1 (make-instance 'Submarine))

(documentation Submarine-Class)          ⇒ "The basic submarine class"
(documentation (class-of Sub-1))         ⇒ "The basic submarine class"
(documentation (find-class 'Submarine)) ⇒ "The basic submarine class"
```

■ :default-initargs

Supplies default values for **:initarg** entries. These entries are almost always ones that were provided by a superclass, since if they were defined in the current class it would be easier to just put in a **:initform** for them. This is an alternative to repeating the slot name and supplying a new **:initform**, but is preferred by some people as being more concise. In addition, in some situations the slot name is kept private, and only the **:initarg**'s are advertized as the official interface. The **:default-initargs** entry overrides any local or inherited **:initform**. The values are evaluated every time **make-instance** is called.

```
(defclass Submarine ()
  ((Depth :accessor Depth :initarg :Depth :initform 100)
   (Speed :accessor Speed :initarg :Speed :initform 5)))

(defclass Slow-Submarine (Submarine)
  ()
  (:default-initargs
     :Speed 1))

(setq Sub-1 (make-instance 'Slow-Submarine))
(Speed Sub-1) ⇒ 1
```

■ :metaclass

Can be used to change the general behavior of classes themselves, by changing the built-in superclass of user-defined classes from the usual default of **standard-class**. Not for the faint of heart.

Exercise: Warm-up

1. Define a Rectangle object with a Width and a Height slot, and a reader and writer for each. Try making a few instances, retrieving and changing slots, etc.

2. Rewrite your Rectangle class, but instead of defining a :reader and :writer for the slots, reproduce their behavior yourself using defmethod.

3. Create a hierarchy of geometric objects (circles, rectangles, squares, and others as desired) similar to the examples shown already. Be able to lookup and change the appropriate fields indicating dimensions for each.

4. Define a method for each that returns the object's area.

5. Create a list of at least 100 objects of various types. Write a function that will take such a list and calculate the sum of the areas.

6. Write functions that will count the number of objects of each class in your list. Summing various entries will add up to greater than the length of your list, since most instances have multiple superclasses. For instance, if you have 100 instances in your list, and your top-level class is Shape, then clearly **`(Number-of-Given-Class 'Shape <List>)`** should return 100.

Common Blunders

■ Omitting parens in arglist in DEFMETHOD.

Writing

```
(defmethod Area (Sq Square) ...)
```

instead of

```
(defmethod Area ((Sq Square)) ...)
```

Lisp will accept the former, and think that you have two unspecialized arguments instead of one argument specialized as a **`Square`**.

■ Missing parens around slot definition list in DEFCLASS.

Writing

```
(defclass Rectangle (Polygon)
  (Width ...)
  (Height ...))
```

instead of

```
(defclass Rectangle (Polygon)
  ((Width ...)
   (Height ...)))
```

Lisp will not accept the former, but the error message is not necessarily clear.

■ Forgetting empty slot definition list if you don't define local slots in DEFCLASS.

Writing

```
(defclass Square (Rectangle))
```

instead of

```
(defclass Square (Rectangle) ())
```

Lisp will not accept the former.

■ Referring to class name instead of instance variable in DEFMETHOD.

Writing

```
(defmethod Area ((Sq Square))
  (* (Width Square) (Width Square)))
```

instead of

```
(defmethod Area ((Sq Square))
  (* (Width Sq) (Width Sq)))
```

Lisp may give a warning about an unknown free variable, but probably won't even do that if you type the **`defmethod`** directly into the Lisp Listener (Lucid doesn't). So you might not get an error until run-time.

■ Forgetting accessors are functions and thus could conflict with built-in function names.

E.g. writing

```
(defclass Graphical-Object ()
  ((Position :accessor Position)))
```

Lisp will not accept this since you cannot redefine the built-in **`position`** function.

■ Putting the new value last instead of first in the definition of a SETF method.

Writing

```
(defmethod (setf Area) ((Sq Square)
                        (New-Area number))
  (setf (Width Sq) (sqrt New-Area)))
```

instead of

```
(defmethod (setf Area) ((New-Area number)
                        (Sq Square))
  (setf (Width Sq) (sqrt New-Area)))
```

Lisp will accept the former, and then users are puzzled as to why **`(setf (Area Square-1) 10)`** doesn't work.

■ Putting the new value last instead of first in a call to a :writer method.

E.g given

```
(defclass Circle ()
  ((Radius :reader Radius
           :writer Set-Radius
           :initform 5)))
(setq Circle-1 (make-instance 'Circle))
```

Writing

```
(Set-Radius Circle-1 10)
```

Instead of

```
(Set-Radius 10 Circle-1)
```

■ Confusion about doc strings in DEFMETHOD.

People often do

```
(defmethod Area ((Rect Rectangle))
  "WIDTH times HEIGHT of the rectangle"
  (* (Width Rect) (Height Rect)))
```

without clearly thinking about what that might mean. Some people think it will make a doc string on the *generic* function that **(documentation 'Area 'function)** or the equivalent emacs keystrokes will retrieve. Others vaguely expect it to make a doc string on each separate method, and that the emacs doc-string retrieving keystroke (which just calls **documentation**) will somehow automagically be able to figure out which method it applies to. In fact, Lisp will accept this, but put the documentation on the method *object*, which beginners probably know nothing about. Use the **:documentation** entry in **defgeneric** to put a doc string on the generic function.

■ Invalid :initargs are accepted by MAKE-INSTANCE.

On SPARC machines, both Lucid Common Lisp 4.1.1 (both the production and development compilers) and Harlequin LispWorks 3.1.0 accept unknown **:initarg**'s without complaint, even at the highest **safety** settings.

```
(defclass Foo ()
 ((Slot-1 :accessor Slot-1 :initarg :Slot-1
:initform 5)))

; typo: SLOT1 instead of SLOT-1
(setq Test (make-instance 'Foo :Slot1 10))
[No error message]

(Slot-1 Test) ==> 5
```

This is a bug in the implementation; all implementations are supposed to flag this as an error.

■ Forgetting the class has to exist before any method that specializes upon it.

Lisp programmers are used to being able to define functions in any order, where even if **Foo** calls **Bar**, **Foo** can be defined first. But

```
(defmethod Area ((Rect Rectangle)) ...)
(defclass Rectangle (Polygon) ...)
```

is illegal. You have to define the class first.

■ Changing a method to apply to a more general class does not supersede previous method.

E.g. a user writes

```
(defmethod Half-Area ((Rect Filled-Rectangle))
 (/ (Area Rect) 2))
```

Then they notice that this functionality could apply to all Rectangles, not just Filled-Rectangles. So they change the class, with their intuition being that they are *replacing* the old definition, when in fact they are *adding* a new, less-specific method. They then later add a call to **float** to avoid getting a ratio back.

```
(defmethod Half-Area ((Rect Rectangle))
  (float (/ (Area Rect) 2)))
```

Then they are puzzled as to why their new definition appears not to have taken effect, since they are testing it on an instance of Filled-Rectangle, which still gets the old, more-specific definition.

Exercise: Setf Methods

1. Arrange it so that you can set either the Width or the Height slot of Squares, but in such a way that they always are guaranteed to have the same value. To the user, these two slots should appear to be synonymous.

```
(setq Square-1 (make-instance 'Square :Width 5))
(Area Square-1) ⇒ 25
(setf (Width Square-1) 6)
(Width Square-1) ⇒ 6
(Height Square-1) ⇒ 6
(Area Square-1) ⇒ 36
(setf (Height Square-1) 7)
(Width Square-1) ⇒ 7
(Height Square-1) ⇒ 7
(Area Square-1) ⇒ 49
```

2. Make it possible to set the Areas of various object types. For some object types (e.g. rectangles), if it seems too arbitrary which dimension to change, feel free to put in explicit messages saying it is not allowed instead.

```
(Area Square-1) ⇒ 49
(setf (Area Square-1) 100)
(Width Square-1) ⇒ 10
```

Simplified Interface: Def-Class

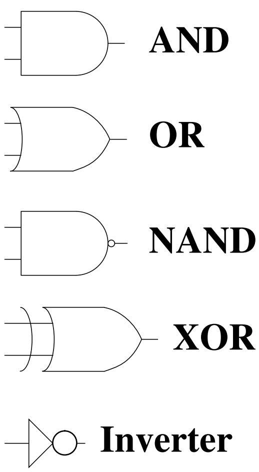
1. Def-Class Macro. (Defined in ~hall/Lisp/CLOS-Utilities.lisp/sbin)

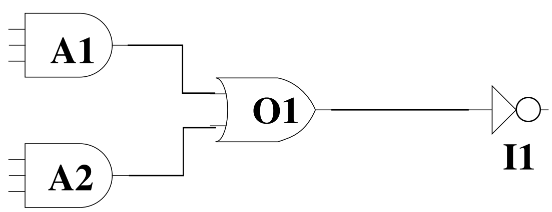
```
(defmacro Define-Class (Class-Name Superclass-List &rest Slot-Descriptions)
  `(defclass ,Class-Name ,Superclass-List
    ,(mapcar #'Expand-Slot-Entry Slot-Descriptions)))

(defun Expand-Slot-Entry (Slot-Description)
  (let ((Name (first Slot-Description))
        (Value (second Slot-Description)) )
    (list
      Name
      :accessor Name
      :initarg (Add-Colon Name)
      :initform value)))

;;; Like (read-from-string (format nil ":~A" Symbol))
(defun Add-Colon (Symbol) (intern (symbol-name Symbol) :keyword))
```

2. IC/Gates Example (see ~hall/Lisp/Gates.lisp)





EXAMPLE

```
(Def-Class Gate ()
  (Input-List '() )
  Operation)

(defmethod Input-Value ((Input number))
  Input)

(defmethod Input-Value ((Input Gate))
  (Output Input))

(defmethod Input-Value-List ((Object Gate))
  (mapcar #'Input-Value (Input-List Object))
)

(defmethod Output ((Object Gate))
  (Boolean-to-Number
    (funcall (Operation Object)
             (Input-Value-List Object))))
```

```
(defun Boolean-to-Number (Boolean)
  "Given NIL returns 0, otherwise returns 1"
  (if
    Boolean
    1
    0) )

(defun All-Ones? (Num-List)
  "Are all entries in Num-List equal to 1?
   This is the operation AND gates use"
  (every #'Equal-One? Num-List))

(defun Equal-One? (Num) (equal Num 1))

(Define-Class And-Gate (Gate)
  (Operation #'All-Ones?))

...
```

Exercise: print-object

1. Define a custom **print-object** method for a couple of your objects. For example, you will note that objects created via **Define-Class** have a **print-object** that includes their **Name** slot and the name of their class in the printout.

An amusing application of **print-object** is to customize your prompt in Lucid Common Lisp. In Lucid, the value of the ***prompt*** variable is intended to be a string that is printed at the beginning of each line. However, Lucid just uses **print** to print it out, and **print** is defined to use the **print-object** method if it prints out CLOS instances. Consequently, if the value of the ***prompt*** variable is a CLOS instance with a custom **print-object** method, you can effectively have the prompt be dynamic, since the **print-object** will get called each time. For instance, in /home/utilities/lisp-init.lisp, there is a function called **Make-Custom-Lisp-Listener-Prompt**, which sets it up so that the prompt contains the name of the current package and a number indicating how many commands you've typed in since entering Lisp. You might want to try something similar.

Here's an example (I call **Make-Custom-Lisp-Listener-Prompt** from my lisp-init.lisp):

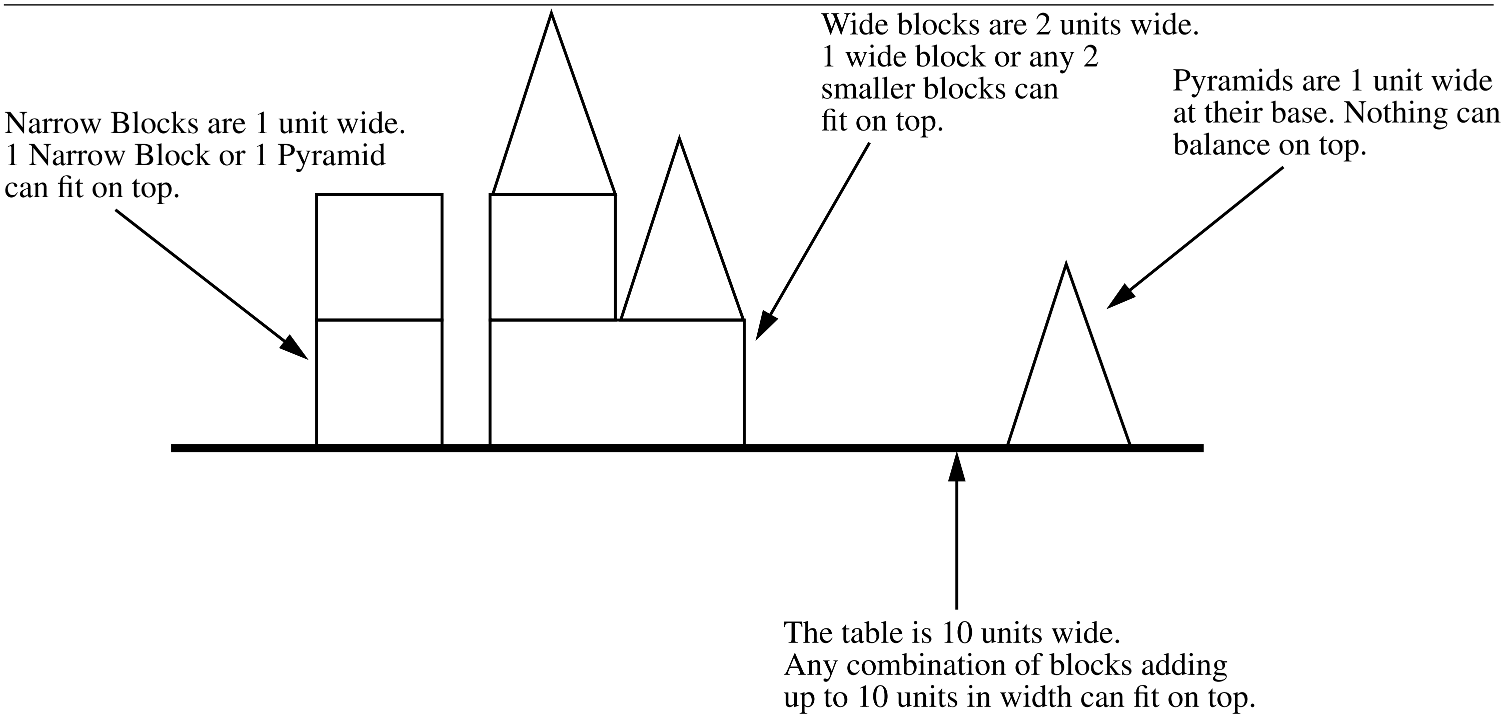
```
CL-USER-1> (+ 2 3)
5
CL-USER-2> (in-package :CLIM-User)
#<Package "CLIM-USER" C9D2C6>
CLIM-USER-3> (+ 2 3)
5
CLIM-USER-4>
```

Intermediate Exercise: Blocks-World

1. Use CLOS to represent information about stacking and unstacking blocks. There are four types of objects: tables, narrow blocks, wide blocks, and pyramids, with information on their characteristics listed below. You should be able to retrieve the following pieces of information about any block: the block's width (at the bottom for pyramids), what block it is on (`Supporting-Block`), what blocks (if any) are directly on top of it (`Supported-Blocks`), what blocks (if any) are anywhere on top of it (`Blocks-Above`), and how much space is still available on top for additional blocks (`Remaining-Space`).

You should have two operators: `Put-On` and `Take-Off`, each of which take two objects as arguments. `(Put-On Object1 Object2)` should, if there is sufficient space on `Object2`, put `Object1` on top. This means that `(Supporting-Block Object1)` should now return `Object2`. `Object1` should now be in the list of supported blocks of `Object2`, and the remaining space of `Object2` should be decremented by the width of `Object1`. `Take-Off` should do the reverse, removing a block and putting it on the table. You can assume that moving a block also moves all blocks that are on top of it, or you could disallow moving a block that has anything on it.

Build a few stacks, showing that the supporting and supported blocks are found correctly, that you can't put blocks on top of ones with insufficient space, etc.



2. If you have time and the inclination, you might want to add Strength and Weight characteristics. You could prohibit putting a block on another if it would make the total weight of any block below it exceed its strength limitation. Or you could have blocks get "crushed" (and maybe the towers they support get toppled) when the total weight on them exceeds the limit.

3. Another more advanced extension would be to add ASCII (or CLIM) graphics to your scenario, so that you could print out the state of the world to see if it matches the information your functions are returning. You could even use :after methods to automatically print the state of the world after every change that you make. If you would like to try my version of this (that uses simple ASCII graphics) while you are doing problems (1) and (2), you can load ~hall/Lisp/Print-Blocks.sbin, and call **`Print-Blocks`** with your table as the argument. This code makes the following assumptions, so it might not work with your code if it doesn't agree with these assumptions:

- There must be classes named **`Wide-Block`**, **`Narrow-Block`**, **`Pyramid`**, and **`Table`**.
- Wide blocks are twice as wide as narrow blocks and pyramids.
- The function to look up the block underneath a given block is called **`Supporting-Block`**.
- The function to look up the list of blocks on top of a given block is called **`Supported-Blocks`**.
- Assuming you have a method called **`Put-On`**, this defines a :after method that automatically prints the configuration after every change. This assumes that the table is stored in the variable **`*Table*`**.

For instance, here is some abbreviated output:

```
CL-USER-7> (load "~hall/Lisp/Blocks.sbin")
CL-USER-8> (load "~hall/Lisp/Print-Blocks.sbin")
CL-USER-9> (setq w1 (make-instance 'wide-block))
+--------------+
|              |
|              |
+--------------+
+-----------------------------------------------------------------------------------+
|                                    TABLE                                          |
|                                                                                   |
+-----------------------------------------------------------------------------------+
#<WIDE-BLOCK-1 (a WIDE-BLOCK)>
CL-USER-10> (setq p1 (make-instance 'pyramid))
   /\   +--------------+
  /  \  |              |
 /    \ |              |
+------++--------------+
+-----------------------------------------------------------------------------------+
|                                    TABLE                                          |
|                                                                                   |
+-----------------------------------------------------------------------------------+
#<PYRAMID-1 (a PYRAMID)>
CL-USER-11> (Put-On p1 w1)
   /\
  /  \
 /    \
+------+
+--------------+
|              |
|              |
+--------------+
+-----------------------------------------------------------------------------------+
|                                    TABLE                                          |
|                                                                                   |
+-----------------------------------------------------------------------------------+
```

Exercise: Multiple Inheritance and Mixins

1. Create a simple hierarchy of polygons, similar to part of your hierarchy from the earlier exercises. Creating Area methods for very much this hierarchy is not feasible. However, regular polygons, in which all side lengths and angles are identical, have a standard formula for Area, where N is the number of sides and L is the length of each side.:

$$Area = \frac{1}{4} N L^2 \cot\left(\frac{\pi}{N}\right)$$

Create a mixin to indicate that a polygon is regular, and then have an Area method that will work on all regular polygons. Create a variety of classes for regular polygons with different numbers of sides, and try calculating the areas for each.

You can test your results by checking the area for regular quadrilaterals (squares) and equilateral triangles ($0.43301L^2$) to see if you get the same results. You could also create a regular polygon with a large number of sides, and see if the area is close to that of a circle of the same circumference. Since you know the length of each side and the number of sides, you know the circumference. If you assume that your polygon is close to a circle, you can then calculate the radius as circumference/2π, and calculate the area of the corresponding circle from there.

Exercise: Timing and Efficiency

One of the common questions with CLOS is how the speed of using objects and methods compares to the speed of alternative representations. See the notes on the second page for hints on timing.

1. Slot access. Compare the efficiency of CLOS slot access as compared to looking up entries from assoc lists (the ORBIS way) and from structures (the other more modern alternative). Create a CLOS class that contains at least 20 slots. Make a list of at least 200 instances of this class.

- Create an assoc list containing the same number of entries as the CLOS class contains slots. Make a list containing the same number of assoc lists as the list of CLOS instances contains. Compare searching the list for an entry with a field that matches a given value using both approaches. Once you've verified that you are correctly finding the entry, you might want to arrange it so that the only match is the last entry (or none at all) so that both methods search the full list. Try it using a field that is near the front of the assoc list, and also with one that is at the end.
- Repeat this process, but this time comparing lookup of structure fields to lookup of CLOS slots.

2. Function call overhead. Compare the efficiency of calling a normal function to a generic function that has many methods defined on it. Also compare the efficiency of the built-in generic function "dispatch" to doing it yourself.

- Create a **`Square`** method, that is defined on numbers as well as on objects of at least 4 other types. Also create a **`Square-Number`** function that does what **`Square`** does on numbers.

```
(defmethod Square ((X number)) (* X X))
(defmethod Square ((Obj Foo)) <Something else>)
(defmethod Square ((Obj Bar)) <Something else>)
(defmethod Square ((Obj Baz)) <Something else>)
...
(defun Square-Number (Num) (* Num Num))
```

  Now, create a list of at least 500 numbers. Compute the sum of the squares both ways. Is there any noticeable difference? What if you add 15 more **`Square`** methods? What if you declare the methods in the opposite order (with the one on numbers being last)?
- Write a **`Square-Method`** function that mimics what the **`Square`** method does. You probably want to use **`typecase`** (or **`type-of`** and **`cond`**) to check the type of the argument and decide what to do. Compute the sum of the squares this way. How does the speed compare to the **`Square`** method? Your answer will depend on where in the **`typecase`** (or **`cond`**) your "method" for numbers is.

3. Instance creation. ORBIS creates most of its objects at load time. However, in some applications (e.g. SMS) you want to create many objects at run-time, and the speed of object creation and initialization is an issue. If you feel so inclined, compare the speed of CLOS instance creation to that of creating structures.

Note: benchmarking is notoriously tricky, and can easily give misleading results. Following are a couple of hints for benchmarking in Lucid Common Lisp:

■ GET-INTERNAL-RUN-TIME has very low fidelity

The typical way to do timing in Common Lisp is either to call the high-level **`time`** macro (for a rough result), or to call **`get-internal-run-time`** before and after (for more accurate numbers). However, for some reason the Lucid clock only ticks in increments of 0.01 seconds, so the difference between subsequent calls to **`get-internal-run-time`** will never be less than 0.01 sec. This is fine for timing long processes, but for these very fast operations, more fidelity is needed. One solution is to call the routine multiple times, time it, and then take the average time. You don't want to do this in an explicit loop, because you are then timing the overhead of the looping as well. So a macro that expands directly into repeated calls is a good solution. Loading ~hall/Lisp/Timing-Macros.sbin will give you a simple timing macro **`Time-Form`** that takes a Lisp form and optionally a number N (defaulting to 20). It executes the form N times, records the total time, and then divides that time by N. E.g. assuming **`Find-Matching-Slot-Value`** is your function from #1,

```
(Time-Form (Find-Matching-Slot-Value Slot-Value Instance-List))
```

■ Garbage collection can throw timing off

Garbage collection (gc) is when Lisp reclaims memory that was allocated but is no longer used. An ephemeral gc, the most common type, takes around 0.1 seconds, and can make your timing results vary. In some sense you want to count this time, since a routine that generates more garbage does slow the system down. But since it is sporadic, you might want to try turning garbage collection off while timing. You can call **`gc-off`** and **`gc-on/egc-on`** yourself, or use t macro, also defined in ~hall/Lisp/Timing-Macros.sbin. E.g.

```
(Without-GC
  (Time-Form (Find-Matching-Slot-Value Slot-Value Instance-List)))
```

■ The compiler optimization settings affect the results

You might want to do your timing runs twice. Once with the default, high error-checking settings, and once with the fast, highly optimized settings. To do the first, type

```
(proclaim '(optimize (safety 3) (compilation-speed 3) (speed 2)))
```

then recompile your files, then rerun them. To do the optimized case

```
(proclaim '(optimize (safety 1) (compilation-speed 0) (speed 3)))
```

then recompile your files and rerun your test cases.

■ The compiler can optimize away some operations.

Consider a function that is used to try to determine the floating-point speed of Lisp:

```
(loop for I from 1 to 100000 do (* 0.1 0.2 0.3 0.4 0.5))
```

With high optimization settings, the compiler will notice that the inner multiplication never gets saved, and will simply remove it, leaving you with the same timing results as if you had done **`(loop for I from 1 to 100000 do NIL)`**

Even if you do

```
(loop for I from 1 to 100000 do
  (setq *Global* (* 0.1 0.2 0.3 0.4 0.5)))
```

the compiler could do the multiplication once outside the loop, or even (if particularly clever), only execute the loop once. The only 100% foolproof method is to **`disassemble`** the code and look at the resultant assembler, but being vigilant for this is half the battle.

■ Use the built-in profiler to find out where a routine spends its time

The built-in metering system will trace a chain of function calls and show the percent of time spent in various subfunctions. It should definitely be used for almost any serious optimization effort. See Chapter 2 of Lucid's Delivery Tool Kit docs for the functions **`start-backtrace-logging`**, **`stop-backtrace-logging`**, and **`summarize-backtrace-logging`**. However, if you wrap your code inside **`With-Metering`** (also defined in ~hall/Lisp/Timing-Macros.sbin) the total time (slightly inflated due to the overhead of timing/metering) will be printed along with the backtrace logging info showing the breakdown of that time.

```
(With-Metering ()
  <Code to Profile>)
```

Defgeneric

:documentation
:method
:method-combination
:argument-precedence-order

■ :documentation

Sets up basic generic function (name, number and names of arguments) plus a documentation string.

```
(defgeneric Area (Geometric-Object)
  (:documentation "Area of geometric object in square meters"))
```

■ :method

Lets you define multiple methods in one location. Use defmethod if you want the methods to be together with the class definition (i.e. all stuff related to rectangles in one place), use defgeneric if you want the methods to be together with each other (i.e. all stuff related to area in one place).

```
(defgeneric Area (Geometric-Object)
  (:documentation "Area of geometric object in square meters")
  (:method ((C Circle)) (* pi (expt (Radius C) 2)))
  (:method ((R Rectangle)) (* (Width R) (Height R)))
  ...)
```

■ :method-combination

Allows you to change how inherited values get returned for a given method. The default is to return a single value that is the value of the most specific method. However, you can specify that the values get combined into a list, added together, etc.
Some options:

STANDARD
The default.

PROGN
Execute all methods that apply in most-specific to least-specific order.

```
(defgeneric Print-Identity (Obj)
  (:method-combination progn))
(defmethod Print-Identity progn ((R Rectangle))
  (print "I'm a Rectangle")
  R)
(defmethod Print-Identity progn ((S Square))
  (print "I'm a Square")
  S)

(Print-Identity Square-1)
"I'm a Square"
"I'm a Rectangle"
⇒ #<Square-1>
```

Other options include **+**, **list**, **and**, **or**, **max**, **min**, **append** and **nconc**.

■ **:argument-precedence-order**

List the args (as specified in the defgeneric parameter list) in the order you want. The default is to consider them in left to right order. For instance, consider:

```
(defmethod Foo (Item (L list))
  (Frobify Item L))

(defmethod Foo ((L list) Item)
  (Munge Item L))
```

Now consider a call to **`Foo`** with two lists: **`(Foo '(A B C) '(D E F))`**. The second version of **`Foo`** will get invoked, since it has the more specific match on the first argument, and the arguments will get munged instead of frobified. However, if you had first done:

```
(defgeneric Foo (Item1 Item2)
  (:documentation "Foo takes 2 entries and Munges or Frobifies them")
  (:argument-precedence-order Item2 Item1))
```

then **`(Foo '(A B C) '(D E F))`** would have gotten the first version, since it is has a more specific match on **`Item2`**, which is listed first in the **`:argument-precedence-order`** entry.

Exercise: Object-Oriented Simulation

This is a more extended exercise to design a simulation based upon CLOS. The domain of your simulation is up to you, but some suggested applications are a submarine scenario, a Star Trek scenario, a hack/rogue like cave-exploration game, or a land battle simulation. The end goal, obviously, is to apply this to ORBIS, but that does not necessarily mean that a submarine simulation is the only (or even best) domain on which to practice the core object-oriented concepts and CLOS constructs. So choose whichever domain you think would be interesting and in which it would be feasible to do something in the time available.

The basic idea is to have instances of several classes of objects that represent entities in your “world”. Each class would have certain behaviors assigned to it. The top-level simulation loop could be quite simple, since the more complex behaviors are assigned to each object individually. For instance, the body of your main loop might look like:

```
(mapc #'Act <Instances>)
```

or maybe

```
(loop for Instance in <Instances> do
  (Act Instance)
  (if (Simulation-Over?)
      (Exit-Simulation)))
```

Different versions of the **`Act`** method might be defined for different classes of objects, or there might be relatively standard versions that make use of other methods that would in turn be specialized on each object type. For instance, **`Act`** might be made up of **`Sense`** and **`Interact`**. **`Sense`** might be a class-specific way of trying to determine who is near to you, making use of, say, **`Location`** slots and instance-specific sensing capabilities. E.g. in a Star-Trek scenario you might say a ship can sense anybody within 10*Current-Sensor-Power-Level units, while an asteroid doesn’t sense anything. Once another object is sensed, the current object could **`Interact`** with it, with the course of action and outcomes of any actions taken to be taken determined by the class of objects, aggressiveness, current strengths and characteristics of each object, etc. You have great freedom in the design; just be sure to make use of the capabilities objects and methods provide. Try to concentrate on the general simulation framework, which will emphasize the OO capabilities, rather than spending all your time implementing some very detailed model for one object.

Two common types of simulation are ones where actions occur only after every user-specified “move” of some main player, or ones that start and run continuously. You may choose either type.

You will need some sort of interface to show what is going on in your simulation. If you’re familiar with CLIM, that might be a good choice. If not, I have put together a very simple CLIM library that lets you create a window and draw text, various geometric objects, and a submarine icon on the window. Information on this library is attached. You could also put together a simple ASCII "graphics" display, or (in a pinch) just use textual output. One suggested approach is to have your main simulation functions update some sort of Location slot in your objects, then have a :before method that erases the object from its old position and a :after method that draws the object in its new position. The library I provided supports a flag to the drawing operations that automatically erases the old position before drawing the new one.

Attached is a copy of the documentation for ~hall/Lisp/Simple-Graphics.lisp (and .sbin). It provides a simple set of drawing functions that can be used as a simple graphical front end to a CLOS simulation. If you use it, I'd suggest that you create a unique Object-ID slot in each object, and then when you draw the object you can pass that Object-ID to the drawing functions, resulting in the old drawing being erased before the new one is drawn.

Here's a sample interaction.