System Construction

Processing Lisp Files

- Forms are processed from top to bottom.
- Definitions will be available for subsequent forms.

Example

- The following doesn’t work when compiled.

  - (defun test ()
      (with-test (print 5)))

    (defmacro with-test (&body body)
      `(progn ,@body))

- Warning: It might work!

Processing Lisp Files

- Processing of Lisp files differs for interpreted and compiled code.

Terminology

- Evaluation: code interpretation & execution of compiled code.
- Interpretation: “One-step” execution.
- Compilation: “Two-step” execution.

1. Semantic analysis and transformation.
2. Execution.

Note on Compilation

- Compilation is typically “just” optimization.
- In Common Lisp, compiled code fulfils certain guarantees.
    - Especially, all macros are fully expanded in compiled code!

Execution Times

- Times at which code can be executed:
    - read time
    - compile time
    - macro expansion time
    - load time
    - run time

Read Time

- Reading: Turning a stream of characters into s-expressions (aka parsing).
- Cf. functions read, read-from-string, etc.

Read-time Execution

- Example:

- (defconstant a 0)
  (defconstant b 1)
  (defconstant c 2)

- (case var
    (#.a (print ...))
    (#.b (print ...))
    (#.c (print ...)))

Read-time Execution

- #. is a dispatching macro character.
- Others: #(#’ #\ #+ #- etc.
- Macro characters: ‘ ` “ ( ) etc.
- Define your own with set-macro-character and set-dispatch-macro-character.
- ...defines code to be executed at read time.

Other dispatching macro characters

- #+ and #- for conditional compilation:
  - #+lispworks ; compile the following form only in LispWorks
  - #-(or acl sbcl) ; don’t compile the following form in Allegro or SBCL
  - #+(or) and #-(and) ; don’t compile the following form
  - *features* contains all the features on which you can conditionally compile

Examples

- (defun open-curl-macro-char (stream char)
    (declare (ignore char))
    (let ((forms (read-delimited-list #\} stream t)))
      `(funcall ,@forms)))

  (setq *readtable* (copy-readtable))
  (set-macro-character #\{ #'open-curl-macro-char)
  (set-macro-character #\} (get-macro-character #\)))

- (set-dispatch-macro-character #\# #\t (lambda (s c n)
                                          (declare (ignore s c n))
                                          ''t))
  (set-dispatch-macro-character #\# #\f (lambda (s c n)
                                          (declare (ignore s c n))
                                          ''nil))

Read-time Execution

- #.(erase-hard-disk)
- Execution of #. forms can be suppressed by setting *read-eval* to nil.
- For further information on read macros, see readtables.

Macro Expansion Time

- That's when macros are expanded. Example:

- (defmacro swap (x y)
  (let ((temp (gensym)))
    `(setf ,temp ,x
           ,y ,temp)))

Run Time

- That’s when code is executed.

Interpreted Code

- Interpreted code consists of:
    - read time
    - macro expansion time
    - run time

Compile Time (file compilation)

- Again, code is processed from top to bottom.
- Each toplevel form is read, macroexpanded and compiled, before proceeding to the next form.
- Loading a compiled file behaves as if the forms are executed from top to bottom.

Compile-Time Execution (file compilation)

- The following doesn't work in compiled code:

- (defun f (x) (+ x x))

```
(defmacro with-test (n &body body)
    (let ((m (f n)))
      `(progn (print ,m) ,@body)))

(defun g ()
   (with-test 5 (print 42)))
```

Compile-Time Execution (file compilation)

- The following distinctions are introduced:
    - compile time: when code is compiled
    - load time: when code is loaded
    - execution time: when code is executed
- It can be specified at which of those times toplevel and non-toplevel code is executed.

Compile-Time Execution (file compilation)

- The following works in compiled code:

- (eval-when (:compile-toplevel)
    (defun f (x) (+ x x)))

  (defmacro with-test (n &body body)
  (let ((m (f n)))
    `(progn (print ,m) ,@body)))

eval-when

- (eval-when (:compile-toplevel :load-toplevel :execute)
    ...)

- :compile-toplevel, :load-toplevel: for toplevel forms

- :execute: for non-toplevel forms

eval-when

- (defmacro defmethod (name args &body)
  `(progn
    (eval-when (:compile-toplevel :load-toplevel :execute)
      ...inform compiler about new method...)
    (add-method (function ,name) ...)))

eval-when

- Compile-time side effects don’t happen in the following case:

- (defun f ()
    (defmethod m (...) ...) ; not toplevel!
    ...)

eval-when

- (eval-when (:compile-toplevel) ...)
  => result isn't available at runtime

- (eval-when (:load-toplevel :execute) ...)
  => default case in file-compiled code

- (eval-when (:compile-toplevel :execute) ...)
  => result is only used during compilation,
  but source code can also be interpreted

eval-when

- Typical use: all three execution times.

- Note: sometimes ‘compile, ‘load and ‘eval are used instead of :compile-toplevel, :load-toplevel and :execute.

load-time-value

- (load-time-value form)
  - executes form once at load time,
    and then uses result as a literal value.
  - may execute many times in interpreted code.
  - doesn’t see the lexical environment!
- (let ((object (load-time-value (make-instance ‘some-class ...))))
  ...)

Summary: Execution Times

- File-compiled code:
  - read time, compile time, macro expansion
    => #., :compile-toplevel
  - load time
    => load-time-value, :load-toplevel
  - run time
    => :execute

Summary: Execution Times

- Interpreted code:
  - read time
    => #.
  - macroexpansion time
  - run time
    => load-time-value, :execute

Packages

- Packages map strings to symbols.

- At read time, strings are interned
  in the “current” package => *package*

- Typically, the default listener has
  common-lisp-user as its current package.

Packages

- Packages have names and possibly nicknames:
  common-lisp-user -> cl-user

- Packages have internal and external symbols.

- Packages can inherit external symbols and import any symbol from other packages.

- A symbol is accessible in a package if it is its home package, or if it is an inherited or imported symbol.

Packages

- intern looks up a string in the current package. If it doesn’t map to a symbol, a new symbol is created in the current package.

- gensym, make-symbol and copy-symbol create symbols that are in no package.

- Any string can map to at most one symbol in one package. The same string can map to different symbols in different packages.

Packages

- some-name => looked up in current package.
- pkg:some-name => must be external in pkg
- pkg::some-name => can be internal in pkg

Standard packages

- common-lisp, cl:
  all predefined operators and variables

- common-lisp-user, cl-user:
  a default “playground”

- keyword: specially treated package for keywords
  (:key, :initarg, :metaclass, etc.)

- keywords always evaluate to themselves.

Defining New Packages

- (defpackage “MINICLOS”
  (:nicknames “MINICL”)
  (:documentation “A minimal CLOS.”)
  (:use “COMMON-LISP”)
  (:shadow “DEFCLASS” “DEFMETHOD” ...)
  (:import-from “UTIL” “WITH-GENSYMS”)
  (:shadowing-import-from ...)
  (:export “DEFCLASS” “DEFMETHOD” ...)
  (:intern “CLASS” “OBJECT” ...)
  (:size 123))

Defining New Packages

- (defpackage :miniclos
  (:nicknames :minicl)
  (:documentation “A minimal CLOS.”)
  (:use :common-lisp)
  (:shadow :defclass :defmethod ...)
  (:import-from :util :with-gensyms)
  (:shadowing-import-from ...)
  (:export :defclass :defmethod ...)
  (:intern :class :object ...)
  (:size 123))

in-package

- (in-package :miniclos)
- Typically specially recognized by IDEs.
- Also useful: list-all-packages, delete-package, unintern.

Packages

- Rules of thumb:
    - Define packages in a separate file,
      and then use in-package in your code.
    - Don’t modify packages on the fly.
    - Don’t make packages too small.

System Definitions

- System definitions <=> make files.
- Not part of ANSI Common Lisp.
- Here: ASDF.

ASDF Example

- (asdf:defsystem :contextl
  :name "ContextL"
  :author "Pascal Costanza"
  :version "0.5"
  :depends-on ("closer-mop" "lw-compat")
  :components
  ((:file "contextl-packages")
   (:file "cx-dynascope"
    :depends-on ("contextl-packages"))
   (:file "cx-special-class"
    :depends-on ("cx-dynascope"))
   ...))

- (asdf:oos ‘asdf:load-op :contextl)

Serial Dependencies

- (asdf:defsystem my-system
  :components ((:file “a”) (:file “b”) (:file “c”))
  :serial t)

- => b depends on a, c depends on b, ...

Benefits of System Definition Facilities

- Systems and files are compiled and loaded in the correct order.

- Especially, if a depends on b, b is compiled and loaded before a is compiled and/or loaded.

- This solves issues for which you otherwise need eval-when.

- See http://www.cliki.net/asdf and http://common-lisp.net/project/asdf-install/tutorial/index.html for more details on asdf.