Common Lisp

a general overview

Overview

- CL history and philosophy
- truth and falsity
- local definitions
- Lisp-1 vs. Lisp-2
- lambda list keywords
- packages, symbols & macros

Overview

- continuations
- dynamic scoping
- iteration vs. recursion
- generalized references
- type system
- execution times

History

- 1958: First mention of the name “LISP”
- 1959: First implementation of LISP
- mid-1970’s: various influential Lisp dialects (including Scheme)
- 1982: “An Overview of Common LISP” (Steele et al.)
- 1984: “Common Lisp the Language” (CLtL, Steele et al.)

CL’s First Goals

- Commonality among Lisp dialects
- Portability for “a broad class of machines”
- Consistency across interpreter & compiler
- Expressiveness based on experience
- Compatibility with previous Lisp dialects
- Efficiency: Possibility to build optimizing compilers
- Stability: Only “slow” changes to the language

CL’s First Non-Goals

- Graphics
- Multiprocessing
- Object-oriented programming

History

- In 1986, ANSI CL standardization started.
  - “a more formal mechanism was needed for managing changes to the language”
  - Substantial changes: loop macro, a pretty printer interface, CLOS, conditions

History

- 1989: “Common Lisp the Language, 2nd Edition” (CLtL2, Steele et al.)
  - “There are now many implementations of Common Lisp [...]. What is more, all the goals [...] have been achieved, most notably that of portability. Moving large bodies of Lisp code from one computer to another is now routine.”
- 1994/95: ANSI Common Lisp
- 1990’s: Common Lisp suffers from “AI Winter”
- Late 2000’s: Common Lisp gains more interest again

Language Philosophies

- Single-paradigm languages: “everything is an X”
  - X = object: Smalltalk, Self, Java, Ruby
  - X = function: Scheme, ML, Haskell
  - X = logic rule: Prolog
- Multi-paradigm languages: use several paradigms in the same language
  - APL, C++, Leda, Common Lisp, PHP, Python

CL as a multiparadigm language

- CL integrates OOP, FP and IP (imperative)
- IP: Assignment, iteration, go.
- FP: Lexical closures, first-class functions.
- IP & FP: Many functions come both with and without side effects:
    - cons & push, adjoin & pushnew,
      remove & delete, reverse & nreverse, etc.

CL Philosophy: OOP

- multiple inheritance
- class & instance variables, initialization & reinitialization
- objects can change their classes at runtime
- classes can change their definitions at runtime
- multi-methods, specialized on classes or single objects
- (user-defined) method combinations
- all important aspects can be configured via the CLOS MOP

CL Philosophy

- Not just a pile of stuff, but all well integrated:
  - All operations are invoked the same way
    (functions, methods, accessors, macros, etc.)
  - All operations have return values
  - Operations can silently change their implementation.
  - Everything is an instance of some class
    and may have methods specialized on it.

Basic data structure: list

- Empty list: () or NIL
- Cons cells: (cons 1 2)
- List = chained cons cells: (cons 1 (cons 2 (cons 3 nil)))
  [recursively defined data structure]
- Short notation: (list 1 2 3)
- [There are also other data structures, like vectors, arrays, structs, objects, files, etc., etc.]

Truth and Falsity

- t and every non-nil value vs. nil
- CL: (cdr (assoc key alist))
- Scheme:

```
(let ((val (assv key alist)))
  (cond ((not (null? val)) (cdr val))
        (else nil)))
```

- Example:

```
(defun member (object list)
  (when list
    (if (eql object (car list))
      list
      (member object (cdr list)))))
```

t and nil

- It’s idiomatic to use t for “most general” and nil for “least general” concepts.
- For example: types t and nil
- Or also: t = standard output stream, nil = standard input stream

Lisp-1 vs. Lisp-2

- In CL, functions and values have different namespaces. In a form,
  - car position corresponds to function space
  - cdr positions correspond to value space
- So you can say

```
(flet ((fun (x) (1+ x)))
  (let ((fun 42))
    (fun fun)))
```

Lisp-1 vs. Lisp-2

- In Scheme, all positions in a form are evaluated the same.
  You can say (((f x) y) z)

- This means: Functions are always lambda expressions that may (or may not) be bound to “normal” variables.

Lisp-1 vs. Lisp-2

- Note: Functions are still first class in CL!
    - look up function objects with:
      (function f) or #'f
    - call functional values as:
      (funcall f 42) or (apply f (list 42))

Lisp-2: Example

- ```
  (defun mapcar (function list)
    (when list
      (cons (funcall function (car list))
            (mapcar function (cdr list)))))
  ```

- (mapcar #’list ‘(1 2 3)) => ‘((1) (2) (3))

- (mapcar (lambda (x) (+ 1 x)) ‘(1 2 3) => ‘(2 3 4)

- (mapcar #’1+ ‘(1 2 3)) => ‘(2 3 4)
  ```

But why Lisp-2?

- Reduced number of accidental name captures.
- Makes defmacro work more reliably.

Lambda Keywords

- &rest, &body: rest parameters
- &optional: optional parameters
- &key, &allow-other-keys: keyword parameters
- &environment picks out the lexical environment
- &aux local variables
- &whole the whole form

Keyword Parameters

- (defun find (item list &key (test #’eql) (key #’identity))
  (when list
    (if (funcall test item (funcall key (car list))
      (car list)
      (find item (cdr list) :test test :key key))))

- (find 2 (list 1 2 3))

- (find “Pascal” *list-of-persons*
          :key #’person-name
          :test #’string=)

Evaluation Orders

- In CL, things are evaluated mostly left to right.
  - specified in all useful cases
  - so (+ i (setf i (+ i 1))) is well-defined.

L2R Rule + Keywords

- ```
  (defun withdraw (...)
    ...)

  ...
    (flet ((withdraw (&rest args
                      &key amount
                      &allow-other-keys)
             (if (> amount 100000)
                 (apply #'withdraw :amount 100000 args)
                 (apply #'withdraw args))))
      ...)
  ...
  ```
  ```

Quoting

- In Lisp, programs and data are represented in the same way.
- (+ 1 2) is a program. (Brussels is a city.)
- ‘(+ 1 2) is a list of three elements. (“Brussels” is a word with eight characters.)
- ‘(+ 1 2) <=> (list ‘+ ‘1 ‘2), or (list ‘+ 1 2)
- This allows Lisp programs to reason about themselves.
  [inspect functions, compute functions, change functions, etc.]

Forms

- Forms are either
  - Variables, for example: x
  - Expressions, for example: (+ x y)
  - Self-evaluating forms, for example: 42 or “Brussels”
- Quoted forms are either
  - Symbols, for example: ‘x
  - Lists, for example: ‘(+ x y)

More on quoting

- The ‘ notation is an abbreviation for QUOTE.
- ‘x <=> (quote x)
- ‘42 <=> (quote 42)
- “x <=> (quote (quote x)) <=> ‘(quote x) [a list with two elements!]

Macros

- Macros are like functions, but they replace forms with new forms rather than compute values.

- Example:
```
(defmacro when (predicate &body implication)
  (list ‘if predicate (cons ‘progn implication)))
```

- (when list (print “this is a list”))
```
<=>
(if list (progn (print “this is a list”)))
```

- [Better definition:
```
(defmacro when (predicate &body implication)
  `(if ,predicate (progn ,@implication))) ]
```

Packages

- Packages and modules are different concepts.
  - (Java screwed this up: In Java, packages are modules...)
- Packages are containers for symbols.
- Symbols can be internal, external or inherited.
- So we don't export functions etc., but symbols!

Packages: How it Works

- When source code is parsed, all (!) languages have to do the following:
  - a string “var” is converted to a symbol ‘var
  - later on, ‘var is mapped to some value
- CL packages map strings to symbols.
- Modules usually map symbols to values.

Packages: How it Works

- (in-package “BANK”)
  (export ‘withdraw)
  (defun withdraw (x) ...)

- Allows other packages to say:
  (bank:withdraw 500) ;; or
  (use-package “BANK”)
  (withdraw 500)

Packages: Why?

- No more name clashes!
- Basic issue in almost all name clash problems:
  How to reconstruct the origin of a name?
- In CL: Don’t lose the origin!
  The same symbol always names the same concept!
- In other words: symbols have identity, while in other languages, names don’t.
- Values are associated with symbols,
  for example: (symbol-function ‘x), (symbol-value ‘y), (find-class ‘person), etc.

Symbols & Macros

- Symbols can be generated at runtime.
- Symbols can be "uninterned" (in no package).
- ```
  (defmacro swap (v1 v2)
     (let ((temp (make-symbol "TEMP")))
       `(let ((,temp ,v1))  ;; no name clashes here!!!
          (setf ,v2 ,v1)
          (setf ,v1 ,temp))))
  ```
  ```

Continuations

- It's possible to return "early" from some code.
- For example:

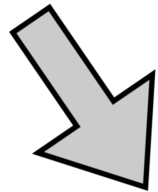
```
(defun find (object list)
  (dolist (x list)
    (when (eql object x)
      (return-from find object))))
```

- No support for "full" continuations.

Dynamic Scoping

- In CL, all global variables are dynamically scoped (”special variables”).
- (Note: not the functions!)
- Dynamic scope: global scope + dynamic extent

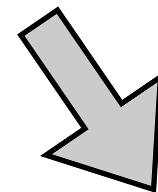
Dynamic Scoping



print-base

do-something

print



Special Variables

- (defvar *class-table*)
- (defvar *class-table* (make-hash-table))
  -> only assign if doesn’t already exist.
- (defparameter *number-of-runs* 20000)
  -> always assign

Recursion vs. Iteration

- Example:

```
(defun find (object list)
  (when list
    (if (eql object (car list))
      object
      (find object (cdr list))))) ;;; this may grow stack space
```

- Better:

```
(defun find (object list)
  (dolist (x list) ;;; this doesn't grow stack space
    (when (eql object x)
      (return-from find object))))
```

setf

- ...or “generalized references”
- like “:=” or “=” in Algol-style languages,
  with arbitrary left-hand sides
- (setf (some-form ...) (some-value ...))
- predefined acceptable forms for left-hand sides
- + framework for user-defined forms

setf

- ```
  (defun make-cell (value) (vector value))

  (defun cell-value (cell) (svref cell 0))

  (defun (setf cell-value) (value cell)
    (setf (svref cell 0) value))
  ```
- `(setf (cell-value some-cell) 42)`
- macros, etc., also supported
  ```

Type System

- CL allows declaration of types
- ```
  (defun add (x y)
      (declare (integer x y))
      (+ x y))
  ```
- CL implementations are not required to recognize them.
- Especially: They must be compatible with dynamic type checking!
  ```

Type System

- Usually, CL implementations take type declarations as a promise for code optimization.
- SBCL and CMUCL do type inferencing and yield useful warnings and even better optimizations.

Execution Times

- CL has well-defined notions of different execution times:
    - read time, compile time, macro expansion time, load time and run time
    - code can be executed at each of those
- also reader macros, compiler macros & “plain” macros,
  but no load-time or run-time macros

Finally

- CL defines a large number of predefined data structures and operations.
  - CLOS, structures, conditions, numerical tower, extensible characters, optionally typed arrays, multidimensional arrays, hash tables, filenames, streams, printer, reader

Important Literature

- Peter Norvig, Paradigms of Artificial Intelligence Programming (PAIP) - CL's SICP
- Paul Graham, On Lisp - *the* book about macros (out of print, but see www.paulgraham.com)
- Peter Seibel, Practical Common Lisp, 4/2005, www.gigamonkeys.com/book/

Important Literature

- Guy Steele, Common Lisp The Language,
  2nd Edition (CLtL2 - pre-ANSI!)
- HyperSpec, (ANSI standard), Google for it!
- My highly opinionated guide, p-cos.net/lisp/guide.html