Common Lisp & Scheme

a comparison

Goals of this Talk

- improve your Common Lisp reading abilities
- understand different “philosophies”
- focus on “incompatible” concepts

Overview

- basic misconceptions
- 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

Basic Misconceptions

- Common Lisp is not dynamically scoped!
- Scheme is not a cleaned-up version of all Lisps!
- Especially, Common Lisp is the newer dialect!
- Steele, Gabriel, “The Evolution of Lisp”, www.dreamsongs.com/Essays.html

History

- 1975: “Scheme - An Interpreter for Extended Lambda Calculus” (Sussman, Steele)

- 1976-1980: ‘Lambda Papers’ (Sussman, Steele)

  - “No amount of language design can force a programmer to write clear programs. [...] The emphasis should not be on eliminating ‘bad’ language constructs, but on discovering or inventing helpful ones.”

History

- 1978: “The Revised Report on Scheme - A Dialect of Lisp” (Steele, Sussman)
  - “It differs from most current dialects of LISP in that it closes all lambda-expressions in the environment of their definition [...], rather than in the execution environment [..., and in] that tail-recursions execute without net growth of the interpreter stack.”

History

- 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

- 1985: "The Revised Revised Report on Scheme or,
  An Uncommon Lisp" (Clinger et al.)
    - "Scheme shares with Common Lisp the goal of a core language common to several implementations. Scheme differs from Common Lisp in its emphasis upon simplicity and function over compatibility with older dialects of Lisp."

History

- 1986: “Revised^3 Report on the Algorithmic Language Scheme” (Rees, Clinger et al.)

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.”

History

- 1991: “Revised^4 Report on the Algorithmic Language Scheme” (Clinger, Rees, et al.)
    - “Programming Languages should be designed not by piling feature on top of feature, but by removing the weaknesses that make additional features appear necessary.”

Further History

- IEEE Scheme (1990)
- ANSI Common Lisp (1994/5)
- ISO ISLISP (1997, mostly a CL subset)
- R5RS (1998, macros now officially supported)
- R6RS in preparation

Scheme Philosophy

- Scheme is a single-paradigm language
    - “everything is a lambda expression”
    - supports mostly functional programming
    - side effects should be marked with a bang!

CL Philosophy

- 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.)
  - Operations can silently change their implementation.
  - Everything is an instance of some class and may have methods specialized on it.

Truth and Falsity

- Scheme: #t and every non-#f value vs. #f
  - predicates end in “?”
- Common Lisp: t and every non-nil value vs. nil
  - predicates usually end in “p” or “-p”

Truth and Falsity

- CL: (cdr (assoc key alist))

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

- “Ballad Dedicated to the Growth of Programs”
  (Google for it!)

Local Definitions

- Scheme:
```
(define (f x)
  (define (g y) (+ x y)) ;; local!
  (g x))
```

- CL:
```
(defun f (x)
  (defun g (y) (+ x y)) ;; not local!!!
  (g x))
```

- So in CL, use let, let*, flet, labels, etc.

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))

But why Lisp-2?!?

- Reduced number of accidental name captures.
- Makes defmacro work more reliably.
- One major difference between Scheme & CL:
    - Either: Lisp-1 is good, macros are a problem.
    - Or: Macros are good, Lisp-1 is a problem.

CL: Lambda Keywords

- CL: (defun f (x &optional y &key test)
  ...)

- Scheme: (define (f . rest)
  ...)

CL: 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

CL: Keyword Parameters

- (defun find (item list &key (test #’eql) (key #’identity))
    ...)

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

Evaluation Orders

- In Scheme, (+ i j k) may be evaluated in any order!
  - this is specified!
  - so never say: (+ i (set! i (+ i 1))) !!!
- In CL, things are evaluated mostly left to right.
  - specified in all useful cases
  - so (+ i (setf i (+ i 1))) is well-defined.

CL: 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))))
    ...)
...
```

CL: Packages

- Packages and modules are different concepts.
  - (Java screwed this up, again: 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! Once and for all!!!
- 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.

CL: Symbols & Macros

- Symbols can be generated at runtime.
- Symbols can be "uninterned" (in no package).
- (defmacro swap (v1 v2)

```
(defmacro swap (v1 v2)
    (let ((temp (make-symbol "TEMP")))
      `(let ((,temp ,v1))  ;; no name clashes here!!!
          (setf ,v2 ,v1)
          (setf ,v1 ,temp))))
```

Continuations

- Short version:
  - Scheme has full continuations.
  - CL only has one-shot escaping continuations.

CL: Dynamic Scoping

- In CL, all global variables are dynamically scoped (”special variables”).
- (Note: not the functions!)
- Dynamic scope: global scope + dynamic extent
- Scheme: Implement it yourself!
  - hard to get right for multiple threads.

CL: 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

Iteration vs. Recursion

- Scheme: Proper tail recursion.
- CL: No requirements, but usually optional tail recursion elimination.
- Scheme: do, named let
- CL: do, do*, dolist, dotimes, loop

CL: 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

CL: 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
  ```

CL: 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!
  ```

CL: 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.

CL: 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
- Apart from these differences, Scheme and Common Lisp are almost the same. ;)

Greenspun’s Tenth Rule

- “Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified bug-ridden slow implementation of half of Common Lisp.”
  - ...probably also true for any sufficiently complicated Scheme program... ;)

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

- ISLISP: www.islisp.info