Advanced Functional Programming, 1DL450

Lecture 5, 2012-11-12

Cons T Åhs

Destructive assignment

- Lisp has “always” had destructive assignment
  - `SET` and `SETQ` used for simple and ordinary destructive assignment.

```
(setq foo '(some list))
```

  - Assigns name in lexical scope or global name (if not in lexical scope)
- `SETF` - generalised destructive assignment
  - If you must have it, make it neat!
  - Simplest case a direct replacement for `SETQ`
  - In general case, you specify the “return value” of a function.
  - Instead of different “functions” for getting and setting, you only have a getter and use it together with `SETF`

Hash tables

- `MAKE-HASH-TABLE` creates a hash table
- `(GETHASH KEY TABLE)` retrieves value from hash table
  - returns two values
    - value or `NIL` if not found
    - `T` or `NIL` to indicate whether value was found
- There is no `SETHASH`! - Instead, combine `SETF` and `GETHASH`

```
CL-USER 233 > (setf *ht* (make-hash-table))
#<EQL Hash Table{0} 200DEE6B>

CL-USER 234 > (gethash 'foo *ht*)
NIL
NIL

CL-USER 235 > (setf (gethash 'foo *ht*) 42)
42

CL-USER 236 > (gethash 'foo *ht*)
42
T
```

SETF is general

```
CL-USER 259 > (setf *list-1* '(1 2 3))
(1 2 3)

CL-USER 260 > (setf *list-2* '(a b c))
(A B C)

CL-USER 261 > (setf *list-3* (append *list-1* *list-2*))
(1 2 3 A B C)

CL-USER 262 > (setf (first *list-1*) 0)
0

CL-USER 263 > *list-1*
(0 2 3)

CL-USER 264 > (setf (rest *list-2*) *list-1*)
(0 2 3)

CL-USER 265 > *list-2*
(A 0 2 3)

CL-USER 266 > *list-3*
(1 2 3 A 0 2 3)
```

SETF is general

- A simple way to destroy shared structures and a lot of nice properties.
- Destructive assignment makes code less readable and can lead to surprises.
- SETF works out of the box for a lot of predefined functions (working on some data), but you can also define your own SETF functions should you want to.
- SETF works on arrays/vectors as well (elements are accessed with `AREF`)
- What about the following?

```
CL-USER 267 > (setf (rest *list-2*) *list-2*)
#1=(A . #1#)
```

Imperative control

- Common Lisp also has a number of control structures that can be used to write more or less imperative program.
- Must be combined with destructive assignment/side effects to be effective.
- `DOLIST` iterates over the elements of a list, but returns `NIL`
- There is also the infamous `LOOP` macro, which can be used to write arbitrarily complex (and more or less unreadable) imperative programs in Lisp.

```
(defun drev (l)
  (let ((rev nil))
    (dolist (x l)
      (setf rev (cons x rev)))
    rev))
```

Macros

- Small core, but very rich in additional constructs
- Most other constructs can be formulated in terms of the core language
  - the core language doesn't grow
  - a program for understanding programs need only understand the core language
  - introduce macros as source code transformation for introducing new concepts in terms of the existing language
- Traditionally (C, Erlang ..) macros are just text replacement performed as preprocessor step before compilation is done.
  - Macros do not increase the expressive power of the language, so they are strictly not needed.
  - This observation has lead to languages excluding macros and suffering the consequences..
- What's so special about Lisp macros?
  - They can execute code at expansion time
  - Functions from code -> code

COND vs IF

- `IF` is all you need for a conditional expression, but gets cumbersome when you want to nest several conditions.
- `COND` is much more convenient.
- Can we define a source code transformation for transforming a `COND` to an equivalent `IF`?
- Since code is represented the same as data, we can reduce this to a function on lists, i.e., transforming the pairs of condition-expression to a nested list.
- Use `REDUCE` - the folding function in Common Lisp.

```
(cond (C0 E0)
      (C1 E1)
       ...
      (Cn En)
      (t E))
```

```
(if C0 E0
    (if C1 E1
        (if
           ...
           (if Cn En
               E)...)))
```

COND vs IF

```
(defun transform-pairs (pairs)
  (reduce #'(lambda (pair code)
              (cons 'if (cons (first pair)
                              (list (second pair)
                                    code))))
          pairs
          :initial-value nil
          :from-end t))
```

```
278 > (transform-pairs
        '(((atom x) (eq x y))
          ((atom y) nil)
          ((equal (car x) (car y)) (equal (cdr x) (cdr y)))
          (t nil)))
(IF (ATOM X) (EQ X Y)
    (IF (ATOM Y) NIL
        (IF (EQUAL (CAR X) (CAR Y))
            (EQUAL (CDR X) (CDR Y))
            (IF T NIL NIL))))
```

- Note that we are actually constructing code inside the function to `reduce`

COND vs IF

```
(defmacro ourcond (&rest pairs)
  (transform-pairs pairs))

(defun ourequal (x y)
  (ourcond ((atom x) (eq x y))
           ((atom y) nil)
           ((ourequal (car x) (car y))
            (ourequal (cdr x) (cdr y)))
           (t nil)))
```

- Use `DEFMACRO` to define a macro.
- The argument `PAIRS` will be bound to a list of the arguments to `OURCOND`.
- The arguments to the macro are *not* evaluated, the "raw" list of arguments are used.
- During compile time, the "call" to `OURCOND` is expanded and the form is replaced with the result (and the compiled).
- The macro must be defined *before* it is used.

LET in terms of LAMBDA

```
(let ((var1 expr1)
      (var2 expr2)
       ...
      (varN exprN))
  body)
```

```
((lambda (var1  var2 ... varN)
   body)
  expr1 expr2 ... exprN)
```

```
(defmacro mylet (bindings &rest body)
  (cons (cons 'lambda (cons (mapcar #'first bindings) body))
        (mapcar #'second bindings)))
```

- We know that we can convert a `LET` to an equivalent application of `LAMBDA`
- `LET` adds readability, but not expressive power
- Again, we are constructing code, but it is a bit difficult to separate the constructed code from the code used to construct it.
- Reader macros to the rescue!

Backquote

- Quote (`'`) returns the argument without evaluation (a special form)
- Backquote (`` ` ``) is similar to quote, but if an element is preceded by
- `,` (comma) the element is evaluated in place
- `,@` (comma-atsign) the result (which should be a list) of evaluating the element is spliced in similar to append

```
CL-USER 292 > `(foo bar)
(FOO BAR)

CL-USER 293 > `(foo ,(+ 1 2))
(FOO 3)

CL-USER 294 > `(foo ,@(list 1 2 3))
(FOO 1 2 3)

CL-USER 295 > (let ((binds '((x (foo z)) (y (bar 2)) (z 3)))
                    (body '(+ x y z)))
                `((lambda ,(mapcar #'first binds) ,body)
                  ,@(mapcar #'second binds)))
((LAMBDA (X Y Z) (+ X Y Z)) (FOO Z) (BAR 2) 3)
```

COND and LET again

```
(defmacro ourcond (&rest pairs)
  (reduce #'(lambda (pair code)
              `(if ,(first pair)
                   ,(second pair)
                 ,code))
          pairs
          :initial-value nil
          :from-end t))

(defmacro mylet (bindings &rest body)
  `((lambda ,(mapcar #'first bindings) ,@body)
    ,@(mapcar #'second bindings)))

(defmacro mylet (bindings &rest body)
  (let ((varnames (mapcar #'first bindings))
        (expresssions (mapcar #'second bindings)))
    `((lambda ,varnames ,@body)
      ,@expresssions)))
```

- The constructed code is easier to see.
- Note the second form of MYLET with local names for the variables and expressions

Adding iteration

- Define a macro `(dolist (var list) body)` which will execute `body` once for each element in the list with `var` bound to the element.
  - Using `MAPCAR` will construct a result, which is wasteful
  - We don't know about `(map nil ..)`
  - Create a local recursive function with `LABELS` to do the job
  - Nice idea, let's try..

Adding iteration

```
(defmacro onlist ((var list) &rest body)
  `(labels ((doit (xs)
              (cond ((null xs) nil)
                    (t ((lambda (,var) ,@body) (first xs))
                       (doit (rest xs))))))
     (doit ,list)))

(defun doit (x) (lose x))

(onlist (e '(1 2 3)) (doit (foo e xs)))
```

```
;; expands to
(LABELS ((DOIT (XS)
           (COND ((NULL XS) NIL)
                 (T ((LAMBDA (E) (DOIT (FOO E XS))) (FIRST XS))
                    (DOIT (REST XS))))))
  (DOIT '(1 2 3)))
```

- Badly named local function since it collides with with a free name in the body
  - We have another collision as well!
- Use `MACROEXPAND-1` and `MACROEXPAND` to see the expansion of a macro
- We need to introduce new unused names - `GENSYM` to the rescue!

GENSYM

- GENSYM will returned a new symbol, belonging to no package.
  - Being new guarantees collision not possible.
  - We thus create the name at expansion time and get a new name for each expansion

```
(defmacro onlist ((var list) &rest body)
  (let ((fname (gensym))
        (lvar (gensym)))
    `(labels ((,fname (,lvar)
                (cond ((null ,lvar) nil)
                      (t ((lambda (,var) ,@body) (first ,lvar))
                         (,fname (rest ,lvar))))))
       (,fname ,list))))

(onlist (e '(1 2 3)) (doit (foo e xs)))

;; expands to
(LABELS ((#2=#:G1021 (#1=#:G1022)
           (COND ((NULL #1#) NIL)
                 (T ((LAMBDA (E) (DOIT (FOO E XS))) (FIRST #1#))
                    (#2# (REST #1#))))))
  (#2# '(1 2 3)))
```

FLET and LABELS

- `FLET` and `LABELS` are used to introduce local function names, similar to `LET`
- Defining these as macros is an interesting and rewarding challenge
  - Use of `LET`, `LAMBDA` and `GENSYM` is allowed
  - Increased understanding of functional programming
  - Increased understanding of macros
  - Increased understanding (and awe!) of Lisp
- Compare with the amount of code needed to do something similar in another language, i.e., extend the language using only the language itself.

Macros

- Backquote makes it very handy to write code that generates code in a clear way
  - More apparent for larger macros
- Always use `GENSYM` when you need to introduce a new name in a macro
  - Without it, there is always a risk of variable capture
- Use macros for actual extension of the language, not just for a shorthand for a function call
  - tempting to use macros as replacement for inline expansion
  - you can't use a macro for a higher order function - why?
- That macros don't distinguish themselves visibly is a mixed blessing
  - code more uniform
  - not apparent what is a macro and extends the language
- A change in a macro leads to all files using the macro having to be recompiled
- Macros can be used to introduce DSLs in a very easy way

Caching revisited

- In Erlang we wrote a cache using the state of a process.
- Another obvious way of writing a cache is to store the cache/state in a variable that survives between function calls
- We want a general solution that can be used for any function
- The state should not be shared between different cached functions
- Define a macro `DEFCACHEFUN` that can be used instead of `DEFUN`
- `LET` can be used surrounding a `DEFUN`
  - A permanent context is created that is accessible in the `DEFUN`

```
(let ((outer nil))
  (defun foo (x y z)
     ... outer ..
     (setf (.. outer) ..)))
```