Modern Lisp 30 lecture — nvlisp and more

Wed, December 30th, 2020 — 2:00AM

Lisp: Complex

- Cons 是所有复杂结构的基本单元，类似 C++ 的 class
- Cons 的 car 和 cdr 分别指向任意类型的对象，不限类型
- Cons 的 car 和 cdr 可以指向 cons 对象，形成复杂结构
- 通过 car 和 cdr 的组合，可以表示任意复杂的数据结构
- 这种设计体现了 Lisp 的灵活性和强大表达能力

在 Lisp 语言中，所有复杂结构都是基于 cons 单元构建的。这种设计体现了 Lisp 的灵活性和强大表达能力。

- 字符串：通过 cons 单元链接字符对象
- 数组：通过 cons 单元链接任意对象
- 哈希表：通过 cons 单元链接键值对
- 函数：通过 cons 单元表示参数列表和函数体
- 宏：通过 cons 单元表示模式匹配和展开规则

这种统一的数据结构设计，使得 Lisp 能够以一致的方式处理各种不同类型的数据，体现了 Lisp 作为一种灵活、强大的编程语言的特点。

```
(defun make-cons (car cdr)
  (lambda (selector)
    (cond ((eq selector :car) car)
          ((eq selector :cdr) cdr)
          (t nil))))

(defun car (cons)
  (funcall cons :car))

(defun cdr (cons)
  (funcall cons :cdr))
```

这段代码实现了 cons 单元的基本功能：

- make-cons 创建一个 cons 单元，接受 car 和 cdr 作为参数
- car 和 cdr 通过调用 cons 单元的相应选择器来获取值

这种实现方式展示了 Lisp 的灵活性，通过闭包和高阶函数来实现数据结构。

```
(defun make-string (&rest chars)
  (reduce #'(lambda (acc c) (cons c acc))
          chars
          :initial-value nil))

(defun string-length (str)
  (cond ((null str) 0)
        (t (+ 1 (string-length (cdr str)))))

(defun string-ref (str n)
  (cond ((null str) nil)
        ((zerop n) (car str))
        (t (string-ref (cdr str) (1- n)))))

(defun string-concat (str1 str2)
  (cond ((null str1) str2)
        (t (cons (car str1) (string-concat (cdr str1) str2)))))

(defun string-reverse (str)
  (reduce #'(lambda (acc c) (cons c acc))
          str
          :initial-value nil))

(defun string-substring (str start &optional end)
  (let ((length (string-length str)))
    (cond ((null end) (setf end length))
          (t nil))
    (cond ((>= start length) nil)
          ((zerop start) (cons (car str) (string-substring (cdr str) 0 (1- end))))
          (t (string-substring (cdr str) (1- start) (1- end)))))

(defun string-equal (str1 str2)
  (cond ((null str1) (null str2))
        ((null str2) nil)
        (t (and (eq (car str1) (car str2))
                (string-equal (cdr str1) (cdr str2)))))

(defun string-compare (str1 str2)
  (cond ((null str1) (if (null str2) 0 -1))
        ((null str2) 1)
        (t (let ((diff (- (car str1) (car str2))))
             (if (zerop diff)
                 (string-compare (cdr str1) (cdr str2))
                 diff))))

(defun string-find (str1 str2 &optional start)
  (let ((length1 (string-length str1))
        (length2 (string-length str2))
        (start (or start 0)))
    (cond ((>= start length1) nil)
          ((>= (+ start length2) length1) nil)
          ((string-equal str2 (string-substring str1 start (+ start length2)))
           start)
          (t (string-find str1 str2 (1+ start))))))

(defun string-replace (str1 str2 str3 &optional start)
  (let ((pos (string-find str1 str2 start)))
    (cond ((null pos) str1)
          (t (let ((before (string-substring str1 0 pos))
                   (after (string-substring str1 (+ pos (string-length str2)) nil))
               (string-concat before (string-concat str3 after)))))))

(defun string-split (str1 str2 &optional limit)
  (let ((length1 (string-length str1))
        (length2 (string-length str2))
        (limit (or limit nil))
        (result nil))
    (cond ((null str2) (cons str1 nil))
          (t (loop for i from 0 to (- length1 length2)
                   do (cond ((string-equal str2 (string-substring str1 i (+ i length2)))
                             (push (string-substring str1 0 i) result)
                             (setf str1 (string-substring str1 (+ i length2) nil))
                             (if (and limit (zerop (1- limit)))
                                 (return-from loop nil)
                                 (decf limit)))
                            (t nil))
                   finally (push str1 result)
                   return (reverse result)))))

(defun string-trim (str1 &optional chars)
  (let ((chars (or chars (make-string #\space #\tab #\newline #\return)))
        (length (string-length str1))
        (start 0)
        (end length))
    (loop for i from 0 to (1- length)
          do (cond ((member (char str1 i) chars)
                    (incf start))
                   (t (return nil)))
          finally (loop for i from (1- length) to 0
                        do (cond ((member (char str1 i) chars)
                                  (decf end))
                                 (t (return nil)))
                        finally (return (substring str1 start end))))))

(defun string-upcase (str1)
  (let ((length (string-length str1))
        (result nil))
    (loop for i from 0 to (1- length)
          do (let ((c (char str1 i)))
               (push (if (lower-case-p c) (upcase c) c) result))
          finally (return (reverse result)))))

(defun string-downcase (str1)
  (let ((length (string-length str1))
        (result nil))
    (loop for i from 0 to (1- length)
          do (let ((c (char str1 i)))
               (push (if (upper-case-p c) (downcase c) c) result))
          finally (return (reverse result)))))

(defun string-capitalize (str1)
  (let ((length (string-length str1))
        (result nil)
        (capitalize-next t))
    (loop for i from 0 to (1- length)
          do (let ((c (char str1 i)))
               (cond ((alpha-char-p c)
                      (if capitalize-next
                          (push (upcase c) result)
                          (push (downcase c) result)
                      (setf capitalize-next nil))
                     (t (push c result)
                        (setf capitalize-next t))))
          finally (return (reverse result)))))

(defun string-trim-left (str1 &optional chars)
  (let ((chars (or chars (make-string #\space #\tab #\newline #\return)))
        (length (string-length str1))
        (start 0))
    (loop for i from 0 to (1- length)
          do (cond ((member (char str1 i) chars)
                    (incf start))
                   (t (return nil)))
          finally (return (substring str1 start nil))))

(defun string-trim-right (str1 &optional chars)
  (let ((chars (or chars (make-string #\space #\tab #\newline #\return)))
        (length (string-length str1))
        (end length))
    (loop for i from (1- length) to 0
          do (cond ((member (char str1 i) chars)
                    (decf end))
                   (t (return nil)))
          finally (return (substring str1 0 end))))
```

这段代码实现了字符串处理的核心功能，包括：

- 基本操作：创建、长度、引用、连接、反转、子串、相等、比较
- 搜索与替换：查找、替换、分割
- 格式化：大小写转换、首字母大写
- 修剪：左侧、右侧、两侧修剪

这些实现展示了 Lisp 在文本处理方面的强大能力，特别是通过递归和高阶函数来处理字符串的灵活性。

Lisp 的 Reader 和 Printer 是语言的核心组件，它们分别负责将文本转换为内部数据结构和将内部数据结构转换为文本。

- Reader：将文本流转换为 Lisp 对象
- Printer：将 Lisp 对象转换为文本表示

这两个组件的设计体现了 Lisp 的灵活性和可扩展性，允许用户自定义数据的输入和输出格式。

```
(defun read-from-string (str)
  (let ((stream (make-string-input-stream str)))
    (read stream)))

(defun print-to-string (obj)
  (let ((stream (make-string-output-stream)))
    (print obj stream)
    (get-output-stream-string stream)))

(defun read (stream)
  (let ((char (read-char stream)))
    (cond ((eq char #\() (read-list stream))
          ((eq char #\") (read-string stream))
          ((eq char #\') (read-quote stream))
          ((eq char #\`) (read-backquote stream))
          ((eq char #\,) (read-comma stream))
          ((eq char #\#) (read-dispatch stream))
          ((digit-char-p char) (read-number stream))
          ((alpha-char-p char) (read-symbol stream))
          (t (error "Unexpected character: ~a" char))))

(defun read-list (stream)
  (let ((result nil))
    (loop
      (let ((char (read-char stream)))
        (cond ((eq char #\)) (return (reverse result))
              (t (unread-char char stream)
                 (push (read stream) result)))))))

(defun read-string (stream)
  (let ((result nil))
    (loop
      (let ((char (read-char stream)))
        (cond ((eq char #\") (return (coerce (reverse result) 'string))
              (t (push char result)))))))

(defun read-quote (stream)
  (list 'quote (read stream)))

(defun read-backquote (stream)
  (list 'backquote (read stream)))

(defun read-comma (stream)
  (let ((char (read-char stream)))
    (cond ((eq char #\@) (list 'comma-at (read stream)))
          (t (unread-char char stream)
             (list 'comma (read stream)))))

(defun read-dispatch (stream)
  (let ((char (read-char stream)))
    (cond ((eq char #\t) t)
          ((eq char #\f) nil)
          (t (error "Unexpected dispatch character: ~a" char))))

(defun read-number (stream)
  (let ((result 0))
    (loop
      (let ((char (read-char stream)))
        (cond ((digit-char-p char)
               (setf result (+ (* result 10) (digit-char-p char))))
              (t (unread-char char stream)
                 (return result)))))))

(defun read-symbol (stream)
  (let ((result nil))
    (loop
      (let ((char (read-char stream)))
        (cond ((alpha-char-p char)
               (push char result))
              (t (unread-char char stream)
                 (return (intern (coerce (reverse result) 'string)))))))

(defun print (obj stream)
  (cond ((null obj) (princ "nil" stream))
        ((t obj) (princ "t" stream))
        ((symbol-p obj) (princ (symbol-name obj) stream))
        ((stringp obj) (princ obj stream))
        ((numbers-p obj) (princ obj stream))
        ((cons-p obj) (princ "(" stream)
                      (print (car obj) stream)
                      (princ " . " stream)
                      (print (cdr obj) stream)
                      (princ ")" stream))
        (t (error "Unprintable object: ~a" obj)))
```

这段代码实现了 Lisp 的 Reader 和 Printer 的核心功能：

- Reader：
  - read-from-string：从字符串读取
  - read：主要的读取函数，根据字符类型调用相应的读取函数
  - read-list：读取列表
  - read-string：读取字符串
  - read-quote：读取引用
  - read-backquote：读取反引号
  - read-comma：读取逗号
  - read-dispatch：处理特殊字符
  - read-number：读取数字
  - read-symbol：读取符号
- Printer：
  - print-to-string：将对象转换为字符串
  - print：主要的打印函数，根据对象类型调用相应的打印函数

这个实现展示了 Lisp 的灵活性和可扩展性，允许用户自定义数据的输入和输出格式。

```
(defun read-from-string (str)
  (let ((stream (make-string-input-stream str)))
    (read
```