Lisp functions used and defined in lectures

Christopher Buckingham

November 14, 2017

Contents

1 The interpreter knows nothing **2**

2 Saving output from your program: Dribble **2**

3 Setf **2**

4 Defining your own functions **3**

5 List processing functions **4**
5.1 cons, append, and list 5
5.2 Assoc 5
5.2.1 Using strings as properties for assoc 6
5.3 Length 6
5.4 Reverse 6

6 Global and local variables **7**
6.1 Let: declares local variables 7
6.2 Typing let directly in the interpreter 7
6.2.1 Let*: changes initialisation options of variables 7

7 dolist **8**

8 Boolean operators **9**
8.1 Member 9

9 The *if* choice function **9**
9.1 Recursive definition of member function 10

10 Arithmetic **11**

11 More on Boolean operators **11**
11.1 Some more predicates 12

12 Choice forms **12**
12.1 When and unless 12
12.2 cond 13

13 Dolist and a recursive alternative **13**
13.1 Answer 13
13.2 Recursive alternative 13

14 Modify extract function so extracts only the first element **14**
14.1 Recursive Lisp solution 14
14.2 Non-recursive using dolist 14

15 *do* template **15**
15.1 Non-recursive extract function using *do* 15

16 Format template **15**
16.1 Format directives 16

17 Reading information **17**
17.1 Read function 17
17.2 Read-line function 17

18 My defined functions for checking yes/no responses **18**
18.1 *elt* function 18

19 Optional parameters **19**

20 Files for input and output **20**

21 Strings and sequences **21**

1 The interpreter knows nothing

Until you type in some commands, there is nothing in the interpreter apart from the common-lisp defined functions. If you want Lisp to know your own functions that you have defined, you can do this two ways:

1. Type the function directly into Lisp
2. Type the function into a file using a text editor and then loading the file into Lisp

If you do it the second way, note that the `load` function call returns the value `T` for True. If your file had, for example, a function call such as `(first '(1 2 3))`, this will evaluate to 1 but not be visible in the interpreter because the interpreter will only return the value associated with the overall function call, which was `load`. It is thus pointless to put such statements in a file unless they are assigning a value to a global variable, which would then be resident in the interpreter. So, for example, `(setf var (first '(1 2 3)))` will set up a global variable called var in the interpreter. If you then type in the name, `var`, to the interpreter, it will return the value of 1. However, the main use for files containing lisp code for loading into the interpreter is to define functions that can then be used. In other words, the file contains your program, which you run by calling the desired functions in the interpreter.

2 Saving output from your program: Dribble

To save a program's input and output as it runs in the interpreter, type `(dribble <filename>)` before calling the function you want to run. All subsequent information displayed on the screen will also be stored in the given file name. To turn it off, just type `(dribble)`.

3 Setf

```
(first '(a b c))
```

- first is a function which returns the first element of a list
- evaluates to *a*.
- If the function was assigned to a variable, the variable would take on the value of *a*.
- *setf* assigns the value of the second argument to the symbol given by the first argument. Stands for SET Field.

```
>(first '(a b c))
A

>(setf var (first '(a b c)))
A

>var
A

>(setf x (* 4 5))

20

>x

20

>(setf x '(* 4 5))

(* 4 5)

>x

(* 4 5)
```

4 Defining your own functions

defun allows you to define your own function.

```
(defun matches (players)
    (- players 1))
```

defun is a function. It is the first member of a list, in the case above, where the list has four members:

first member Function call (to define one)

second member Function name

third member List of parameters

fourth member A form (i.e. an s-expression for evaluation).

As follows:

```
(defun matches         ;; function name
    (players)          ;; parameters
  (- players 1))       ;; form defining function steps
```

The general case is given by:

```
(defun <function name>
       (<parameter 1>..<parameter n>) ;; many parameters
       <form 1>...<form n>)           ;; many forms
```

Value is the last step in the evaluation of the form.

5 List processing functions

```
>(setf l'(1 2 3))
(1 2 3)

>(third l)
3

>(setf l '(1 2 3))
(1 2 3)

>l
(1 2 3)

>(first l)
1

>(last l)
(3)

>(second l)
2

>(third l)
3

>(rest l)
(2 3)

>(rest '(1))
NIL

>(first nil)
NIL

>(rest nil)
NIL
```

5.1 cons, append, and list

cons returns a list with the first argument at the head of the list represented by the second argument:

```
>(cons 1 '(2 3))
(1 2 3)
```

Short for **CONS**truct.

append combines arguments that are lists into a single list:

```
>(append '(1 2 3) '(4 5 6) '(7 8 9) '(Whehey))
(1 2 3 4 5 6 7 8 9 WHEHEY)
```

list makes a list out of its arguments that can be anything:

```
>(list 1 2 3 '(a b) 'c)
(1 2 3 (A B) C)
```

None of these functions change their arguments.

Note that setf can bind more than one variable at once in pairs:

```
>(setf list-1 '(1 2 3) list-2 '(4 5 6))
(4 5 6)

>(setf list-3 (append list-1 list-2))
(1 2 3 4 5 6)

>list-1
(1 2 3)

>list-2
(4 5 6)

>list-3
(1 2 3 4 5 6)
```

To create new list by appending two old lists, use *setf*.

5.2 Assoc

Operates on “association lists”: lists of lists with each member list having only two elements.

```
>(setf jagger '((occupation rock-star)
                (age 58) (sex male)))
((OCCUPATION ROCK-STAR) (AGE 58) (SEX MALE))
```

assoc retrieves any sub-list using the first member as the key:

```
>(assoc 'age jagger)
(AGE 58)
```

General operation is as follows:

```
(ASSOC <key> <association list>)
```

5.2.1 Using strings as properties for assoc

You might want to have an association list with a string as the property if, for example, it was the name of something, with the values following. Take the horse-racing example:

```
(setf horses
 '(HORSES (("pegwell bay"
           ((FORM ((LAST-RACE 8) (SECOND-LAST 1) (THIRD-LAST 1)))
            (FITNESS ((DAYS-SINCE-LAST-RACE 34) (RACES-IN-SEASON 9)))
            (TRAINER ((NAME BAILEY) (STRIKE-RATE 12)))))
          ("red rum"
            ((FORM ((LAST-RACE 1) (SECOND-LAST 8)(THIRD-LAST 3)))
             (FITNESS ((DAYS-SINCE-LAST-RACE 42)(RACES-IN-SEASON 4)))
             (TRAINER ((NAME GIFFORD) (STRIKE-RATE 19)))))
          ("dessie"
            ((FORM ((LAST-RACE 1) (SECOND-LAST 1) (THIRD-LAST 1)))
             (FITNESS ((DAYS-SINCE-LAST-RACE 21) (RACES-IN-SEASON 5)))
             (TRAINER ((NAME ELSWORTH) (STRIKE-RATE 23))))))))))
```

If you want to access a particular horse, using assoc in the normal way will not work:

```
> (assoc "red rum" (second horses))
NIL
```

So you need to use the :test keyword to let assoc know that a different form of equality is required and then to provide the function for that equality, as was done for the `member` function.

```
> (assoc "red rum" (second horses) :test #'string=)
("red rum"
 ((FORM ((LAST-RACE 1)(SECOND-LAST 8)(THIRD-LAST 3)))
  (FITNESS ((DAYS-SINCE-LAST-RACE 42) (RACES-IN-SEASON 4)))
  (TRAINER ((NAME GIFFORD) (STRIKE-RATE 19)))))
```

5.3 Length

Returns the length of a list:

```
>(length list-3)
6
```

5.4 Reverse

Have a guess:

```
>(reverse list-3)
(6 5 4 3 2 1)

>(setf l '(1 2 (3 4 5) 6 7 (8 9) 10))
(1 2 (3 4 5) 6 7 (8 9) 10)

>(reverse l)
(10 (8 9) 7 6 (3 4 5) 2 1)
```

6 Global and local variables

- Global variables remain in the lisp environment after a program has finished executing.
- Distinguished them from local ones by asterisk at start of name
- *<global-variable>
- use **as rarely as possible**.

```
>(setf global-by-mistake 999)
999

>global-by-mistake
999
```

will set up *global-by-mistake* as a permanent variable
Typing name into interpreter will evaluate globals.

6.1 Let: declares local variables

```
(let ((<parameter 1><initial value 1>)
      ...
      (<parameter n><initial value n>))
  <form 1>
  ...
  <form n>)
```

6.2 Typing let directly in the interpreter

Using "let" stops the variable persisting:

```
>(let ((var 999))
   (format t "Variable var has value ~a~%"))

Variable var has value 999
NIL

>var

Error: The variable VAR is unbound.
Error signalled by EVAL.
Broken at EVAL.  Type :H for Help.
```

6.2.1 Let*: changes initialisation options of variables

```
(let* ((<parameter 1><initial value 1>)
       ...
       (<parameter n><initial value n>))
  <form 1>
  ...
  <form n>)
```

- With `let`, parameter n cannot be assigned an initial value given to an earlier variable when it is declared. Setting parameter n to parameter 1, say, during the declarations would cause an error.
- `let*` works exactly the same way as `let` but does allow one to use an earlier parameter as the initial value of a later parameter during decleration.

```
[2]> (let ((v1 2)(v2 4)) (format t "~%v1 = ~a and v2 = ~a~%" v1 v2))

v1 = 2 and v2 = 4
NIL

[3]> (let ((v1 2)(v2 v1)) (format t "~%v1 = ~a and v2 = ~a~%" v1 v2))

*** - EVAL: variable V1 has no value
The following restarts are available:
USE-VALUE      :R1      You may input a value to be used instead of V1.
STORE-VALUE    :R2      You may input a new value for V1.
ABORT          :R3      ABORT
Break 1 [4]> :q

[5]> (let* ((v1 2)(v2 v1)) (format t "~%v1 = ~a and v2 = ~a~%" v1 v2))

v1 = 2 and v2 = 2
NIL
```

7 dolist

Allows you to apply operations to each member of the list.

```
;;
;;
(defun double (l)
  (let ((double-list nil))
    (dolist (element l double-list)
            (setf double-list
                  (cons (* 2 element) double-list)))))
```

Note:

1. element is the variable name given to each member of the list as the list is processed
2. l is the list being processed
3. double-list is the result to be returned by the dolist function
4. If there is no result form, then the dolist function returns nil

```
>(double '(1 2 3))
(6 4 2)
```

Returns nil if third parameter (i.e. double-list) omitted:
dolist (element l)
Dolist and therefore the function returns nil.

8 Boolean operators

Equality operators (Table adapted from Winston & Horn).

equal	true if arguments are the same expression
eql	true if arguments are the same symbol or number
eq	true if arguments are the same symbol (identical memory chunk)
=	true if arguments are the same number

Table 1: Equality predicates

Each tries test below before testing itself
Because computationally more expensive the higher up the table

1. “equal” tries “eql” and then itself
2. “eql” first tries ‘”eq” and then itself
3. etc.

8.1 Member

Returns the rest of the list with the element at the head.

```
>(member 't '(i n t e l l i g e n t))
(T E L L I G E N T)
```

In Lisp, nil is false and anything else is taken to be true.

```
>(equal (* 4 2) 8)
T

>(equal (* 4 2) 9)
NIL

>(member 't '(i n t e l l i g e n t))
(T E L L I G E N T)

>(member 'u '(i n t e l l i g e n t))
NIL
```

9 The *if* choice function

```
(if <test> <then form> <else form>)

(defun biggest (x y)
  (if (> x y)
      x
    y))

>(biggest 3 5)
```

```
5

>(biggest 5 3)
5
```

Note: only one form for the if part and one form for the else part.
To carry out multiple steps, need a function which allows one form to consist of many subforms.
Need *progn* which can contain any number of forms:

```
(defun bigger (x y)
  (if (> x y)
      (format t "~%first argument is bigger")
    (progn (format t "~%second argument is bigger")
           (format t "~%here is second form
                          for else bit~%"))))

>(bigger 3 4)

second argument is bigger
and here is a second form for the else bit
NIL
```

progn returns the value of its **last** form
prog1 returns the value of its first form.

```
(defun bigger (x y)
  (if (> x y)
      (progn (format t "~%first argument is bigger")
             1)
    (progn (format t "~%second argument is bigger")
           (format t "~%and here is a second form
                      for else bit~%"))))

;; now call it in lisp

>(bigger 4 3)

first argument is bigger
1
```

Returns 1, value of second form in the if part.

9.1 Recursive definition of member function

```
(defun member1 (item l)
  (unless (null l)
    (if (equal item (first l))
        l
      (member1 item (rest l)))))
```

10 Arithmetic

Three main types of data structure: integer, ratios, and floating point.

Division is the main function distinguishing them:

```
>(/ 4 2)   ;; integer division
2

>(/ 5 2)   ;; ratio number returned
5/2

>(/ 5 2.0) ;; floating point returned
2.5
```

The last result is due to *floating contagion*

round returns two results: the nearest integer and the remainder

```
>(round (/ 7 3))
2
1/3

;; default situation assigns first result

>(setf n (round (/ 7 3)))
2
```

There are ways of obtaining any or all of the values from multiple-valued forms.

round returns the nearest **even** whole number if the division is equally between two integers.

max Maximum of its numerical arguments

min No idea what this does

sqrt Or this

expt seen it before

abs Absolute value

11 More on Boolean operators

Member tests for equality using eql.

Member can be asked to use a different equality operator

What is returned by the following?

```
>(member '(a 1) '((a 1) (b 2) (c 3) (d 4) (e 5))
         :TEST #'EQUAL)
```

```
((A 1) (B 2) (C 3) (D 4) (E 5))
```

- Any symbol beginning with a colon is a keyword
- Keywords always evaluate to themselves
- Argument following the keyword is the argument for that keyword.
- The procedure to use is the one associated with *equal*.

The same procedure can have different keywords

```
(member '(a 1) '((a 1) (b 2) (c 3) (d 4) (e 5))
          :TEST-NOT #'EQUAL)
((B 2) (C 3) (D 4) (E 5))
```

11.1 Some more predicates

atom	true if argument is an atom
numberp	true if argument is a number
symbolp	true if argument is a symbol
listp	no idea
endp	true if empty list (error if not list)
zerop	ummmmm, give me a minute
plusp	true if argument is positive
minusp	true if argument is negative
evenp	true if argument is even
oddp	true if argument is wierd
$>$	true if arguments in descending order
$<$	true if arguments in ascending order
and	true if all arguments true; returns value of last argument
or	true if any argument true; returns value of first non-nil argument
not	evaluates to true if argument is not true.

Table 2: More predicates

12 Choice forms

The *if* choice done before.

12.1 When and unless

```
(when <test> <then forms>)

(unless <test> <else forms>)
```

Both *when* and *unless* can have any number of forms.
They return the value given by the last one.

12.2 cond

```
(cond (<test 1> <consequent 1-1> ...)
      (<test 2> <consequent 2-1> ...)
         .
         .
         etc
         .
      (<test n> <consequent n-1> ...))
```

The first test to evaluate to true has consequents executed.
Any number of consequents for each test
Final consequent is value returned.
If none of the tests evaluate to true, then cond returns nil

13 Dolist and a recursive alternative

Write a function called *extract* which uses dolist to remove all of the elements from the list:

```
>(extract 'n '(i n t e l l i g e n c e))
(I T E L L I G E C E)
```

13.1 Answer

```
;; returns a list with all the elements removed
;;
(defun extract (element l)
  (let ((new-list nil))
    (dolist (item l (reverse new-list))
            (unless (eql item element)
              (setf new-list (cons item new-list))))))
```

13.2 Recursive alternative

English description:
  Unless the list is empty

1. If the item is equal to the first member of the list, return the rest of the list with the item also extracted from it;
2. otherwise add the first member of the list to the rest of the list where the item has been extracted.

```
;; Extracts all members of the list recursively
;;
(defun extract (item l)
  (unless (null l)
    (if (equal item (first l))
        (extract item (rest l))
      ;; else add first item to accumulating list
      (cons (first l) (extract item (rest l))))))
```

14 Modify extract function so extracts only the first element

Write a function called *extract* which uses dolist to remove the first element from the list:

```
>(extract 'n '(i n t e l l i g e n c e))
(I T E L L I G E N C E)
```

Write down the solution in English

```
If element equals first member of the list
    return the rest of the list
Else
    return list with first element at head
    and rest of list with first element extracted
```

14.1 Recursive Lisp solution

```
;; returns list with first occurrence of element removed
;;
(defun extract (element l)
  (unless (null l)
    (if (eql element (first l))
        (rest l) ;; return the rest of the list
      ;; else return list with first element at head
      ;; and rest of list with first element extracted
      (cons (first l) (extract element (rest l))))))


>(extract t '(i n t e l l i g e n t))
(I N E L L I G E N T)
```

14.2 Non-recursive using dolist

```
;; extracts the element from the list
;; EFFICIENT because uses the return call
;; when the element is found.
;;
(defun extract (element l)
  (let ((new-list nil))
    (format t "~% efficient dolist~%")
    (dolist (item l (reverse new-list))
      (if (eql item element)

          ;; return list without element by appending
          ;; new-list to list starting after the element
          ;; (member returns this part of the list but
          ;; also including the undesired element)
          (return (append (reverse new-list)
                          (rest (member element l))))
        (setf new-list (cons item new-list))))))
```

15 *do* template

```
(DO ((<parameter 1><initial value 1><update form 1>)
     (<parameter 2><initial value 2><update form 2>)
     ...
     (<parameter n><initial value n><update form n>))
    (<termination test><intermediate forms, if any>
     <result form>)
   <body>)
```

15.1 Non-recursive extract function using *do*

The function *do* is more general than dolist but harder to use. Here it is used to define the extract function.

```
;; returns list with the first occurrence of element removed
;;
(defun extract (element l)
  (do ((new-list nil)          ;; parameter
       (item (first l)))       ;; parameter
      ;; condition for exiting from do
      ((or (eql item element) (endp l))
       ;; result to return (leaving off item from l)
       (append (reverse new-list) (rest l)))
      ;; What to do on each iteration
      (setf new-list (cons item new-list)
            l (rest l))
      ;; set up next item of list
      (setf item (first (rest l)))))
```

16 Format template

```
(format destination control-string arguments)
```

- format is used to produce formatted output.
- Outputs the string but display is modified by directives.
- Directives prefixed by ˜
- Most directives take the next argument in the list following the string and format its contents in a particular way.
- Errors occur if a directive needs an argument but there are none left.
- The output goes to destination:

  t to the terminal (i.e. the screen)
  filename to the named file
  nil to a string, returned as the format function's value.

```
[4]> (format t "hello")
hello
NIL
```

```
[5]> (format nil "hello")
"hello"

[6]> (setf var (format t "hello"))
hello
NIL

[7]> var
NIL

[8]> (setf var (format nil "hello"))
"hello"

[9]> var
"hello"
```

16.1 Format directives

Suppose v1, v2, and v3 are variables with the values of 1, 2, and 3 respectively.

~ A Prints the argument verbatim. Use @ for padding.

```
>(setf v1 1 v2 2 v3 3)
3

>(format t
"Print no padding:~A;
padding to left:~8@A;
padding to the right:~8A lovely" v1 v2 v3)

Print no padding:1;
              padding to left:       2;
              padding to the right:3        lovely
NIL
```

~ S Prints strings with inverted commas. If v4 is a variable with the value “this is a string”, then the following happens:

```
>(format t "~s" v4)
"this is a string"
NIL

>(format t "~a" v4)
this is a string
NIL
```

~% Breaks the line

```
>(format t "break line here~%and carry on")
break line here
and carry on
NIL
```

~F Prints floating point numbers. It takes a number of additional arguments:

w total width of representation

d number of digits after decimal point

```
(setf lisp-programmer 45132.34)

>(format t
"~%Number is ~F or like this: ~10,1F or like this ~,3F~%"
lisp-programmer lisp-programmer lisp-programmer)

Number is 45132.34

or like this:    45132.3 or like this 45132.340
NIL

>
```

17 Reading information

17.1 Read function

To get information and put it in a variable, the `read` function is used:

```
> (setf number-of-rooms (read))
4
4

> number-of-rooms
4
```

- No white spaces in input
- Errors otherwise

17.2 Read-line function

This one reads a string, ignoring the end-of-line character:

```
>(setf var (read-line))This is a line of text
"This is a line of text "

>var
"This is a line of text "
```

18 My defined functions for checking yes/no responses

Uses a useful function called `elt`

18.1 *elt* function

`(elt <sequence> number)`
returns the element which is the number away from the first element, counting from zero:

```
>(elt "programmer" 2)
#\o

>(elt '(a b c) 2)
C

>(elt "programmer" 0)
#\p
```

- Returns the element of the sequence in the appropriate form.
- Strings are sequences of characters and characters are printed as `#\c`.

```
;;  This function reads the answer to a question and
;;  returns TRUE if it is "Y" or "y" and false otherwise.
;;
(defun affirmative ()
  (format t
"~%Type `Y' followed by <RETURN> for YES or 'N' and <RETURN> for NO.~%")
  (let ((response (read-line)))
    (cond ((zerop (length response)) ;; nothing entered
           (format t "
WARNING!!! You have pressed <RETURN> on its own.
Please answer again.~%")
           (affirmative)) ;; recursively call affirmative
          ;; check first letters: allows a user to enter
          ;; the whole word (yes or no) as well as y or n.
          ((char-equal #\y (elt response 0))
            T)
           ((char-equal #\n (elt response 0))
            nil)
           (t
            (format t "
WARNING!!! You have not entered either a 'Y' or 'N'.
Please answer again.~%")
            (affirmative)))))

(defun test-affirmative ()
  ;; to read the carriage return after entering function call
  (read-line)
  ;; now test affirmative
  (affirmative))
```

```
;;  Reads a user-supplied symbol, prints it, and asks
;;  whether symbol correctly entered. If yes, returns it.
;;
(defun read-symbol ()
  (format t
"~%Enter the name followed by <RETURN>.
Remember that the name must be all one word or else hyphenated~%")
  (let ((name (read)))
    (format t
"~%The name you have entered is ~a. Is it correct?~%" name)
    (if (affirmative)
        name
      (read-symbol))))
```

19 Optional parameters

Signified by the &optional parameter.

```
(defun opt (&optional n1 n2 n3)
  (if (and n1 n2 n3)
      (list n1 n2 n3)
    (if (and n1 n2)
        (list n1 n2)
      (if n1
          (list n1)
        (format t "~%No arguments given~%")))))
```

All arguments following the &optional parameter signify that the arguments can be there or not.

```
>(opt)

No arguments given
NIL

>(opt 1)
(1)

>(opt 1 2)
(1 2)

>(opt 1 2 3)
(1 2 3)

>(opt 1 2 3 4)

Error: OPT [or a callee] requires less than four arguments.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signalled by OPT.
Broken at OPT.  Type :H for Help.
>>
```

20 Files for input and output

This text is taken from the lecture showing how mind-maps can be converted into Lisp lists from the underlying XML.

- Very simple to use files in Lisp with a function called `with-open-file`.
- Need to tell the file whether it should be input (i.e. read from) or output (i.e. written to). It works as follows:

```
(with-open-file
  <internal file name> <external-file-name>
  :direction :input)
```

or

```
(with-open-file
  <internal file name> <external-file-name>
  :direction :output)
```

For example, to read from the house XML file, we would do the following:

```
(defun test-file (filename)
  (with-open-file
   (house-xml filename :direction :input)
   (let ((s nil))
     (setf s (read-line house-xml nil 'eof))
     (format t "~a~%" s))))

;; Check it
;;
> (test-file "house-template.mm")
<map version="0.8.0">
NIL
```

So it works. And if we wanted to write to a file, we simply change the direction from input to output.

```
(defun test-write (filename)
  (with-open-file
   (house-xml filename :direction :output)
   (let ((s nil))
     (format t "~%write some XML~%")
     (setf s (read-line))
     (format house-xml "~a~%" s))))

> (test-write "test-output-file.txt")

write some XML
<THIS IS AN XML NODE THAT AIN'T REALLY XML>
NIL
```

1. Now lets view the output file in Emacs:
2. `<THIS IS AN XML NODE THAT AIN'T REALLY XML>`

21 Strings and sequences

- The program we need to write uses Lisp string and sequence functions.
- Strings are just specialised sequences, so anything that works on sequences will also work on strings. Sequences are just ordered objects, so any list is also a sequence.
- The string and sequence functions we are interested in are:

1. `(search seq1 seq2)` that returns the position (zero-indexed) of the first subsequence of sequence2 that matches sequence 1.
   (a) Takes keywords: start2 and end2 gives the start and end position of seq2 that needs to be searched.
   (b) `(search "TEXT" nod :start2 7)` searches nod from position 7.
2. `(subseq seq start & optional end)` that returns the subsequence of seq starting at position start and ending at position end, or the end of the string if no end given. Start and end are numbers, with the first place in the string given a 0.
3. `(subseq nod 6 15)` returns the subsequence of nod from position 6 to 15

```
> nod
"<node TEXT=\"rooms\" POSITION=\"left\">"

> (search "TEXT" nod)
6

> (search "TEXT" nod :start2 7)
NIL

> (subseq nod 6 15)
"TEXT=\"roo"
```