Symbols and S-Expression Trees

SOLUTIONS

CS251 Programming Languages
Spring 2019, Lyn Turbak

Department of Computer Science
Wellesley College

Paul Graham's *Revenge of the Nerds*

What made Lisp different

6. **Programs composed of expressions.** Lisp programs are trees of expressions, each of which returns a value. …

7. **A symbol type.** Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.

8. **A notation for code using trees of symbols and constants.** *[Lyn adds: these trees are called **symbolic expressions** = **s-expressions**]*

9. **The whole language there all the time.** There is no real distinction between read-time, compile-time, and runtime. … reading at runtime enables programs to communicate using s-expressions, an idea recently reinvented as XML. *[Lyn adds: and JSON!]*

Symbols

Lisp was invented to do **symbolic processing. (**This was thought to be the core of Artificial Intelligence, and distinguished Lisp from Fortran (the other main language at the time), whose strength with **numerical processing**.)

A key Racket value is the **symbol**.

The symbol `cat` is written `(quote cat)` or `'cat`.

Symbols are values and so evaluate to themselves.

```
> 'cat
'cat

; 'thing is just an abbreviation for (quote thing)
> (quote cat)
'cat
```

Symbols are similar to strings, except they're **atomic**; we don't do character manipulations on them.

Testing Symbols for Equality: `eq?`

The key thing we do with symbols is test them for equality with `eq?` (pronounced "eek"). A symbol is `eq?` to itself and nothing else.

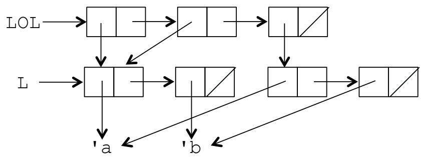
```
> (eq? 'cat 'cat)
#t
> (map (λ (s) (eq? s 'to))
       (list 'to 'be 'or 'not 'to 'be))
'(#t #f #f #f #t #f)
```

`eq?` on Symbols and Lists

`eq?` can be used on any Racket values. It is used to test if two values are the **same object** in memory.

In contrast, `equal?` tests **structural equality** of two values.

```
(define L (list 'a 'b))
(define LOL
  (list L L (list 'a 'b)))
```



```
> (eq? (first L) (second L))
#f
> (eq? (first L) (first (third LOL)))
#t
> (eq? (first LOL) (second LOL))
#t
> (eq? (first LOL) (third LOL))
#f
> (equal? (first LOL) (third LOL))
#t
```

More `eq?` examples

```
> (eq? "cat" "cat")
#t ; Happens to be true, but not guaranteed

> (eq? "cat" (string-append "c" "at"))
#f ; Two strings with the same chars not guaranteed eq?

> (equal? "cat" (string-append "c" "at"))
#t ; Two strings with the same chars guaranteed equal?

> (eq? (fact 5) (fact 5))
#t ; For “small” numbers, eq? is same as =

> (eq? (fact 1000) (fact 1000))
#f ; = bignums are not guaranteed eq?, but are equal?

> (eq? 'cat (string->symbol "cat"))
#t ; string->symbol returns unique symbol for a string

> (eq? (string->symbol "cat")
       (string->symbol (string-append "c" "at")))
#t ; only one symbol in memory with a given name
```

Quotation with Lists

As you’ve seen, a single quote can be used with parenthesized structures to denote lists.

You can think of `'(to be or not to be)` as a sugared form of `(list 'to 'be 'or 'not 'to 'be)`. (Not quite true, but useful.)

A quoted parenthesized structure `(quote (...))` (abbreviated `'(...)`) denotes a list, according to the following desugaring:

`(quote (`*`thing_1`* … *`thing_n`*`))`
desugars to `(list (quote `*`thing_1`*`)` … `(quote `*`thing_n`*`))`

Quoted Atoms

Atomic (indivisible) elements that can appear in list structures areare called ***atoms***. In Racket, atoms include numbers, booleans, and strings in addition to symbols.

```
(define (atom? x)
  (or (number? x) (boolean? x)
      (string? x) (symbol? x))))
```

A quoted atom `(quote `*`atom`*`)` (abbreviated `'`*`atom`*) denotes the atom. For atoms that are not symbols, `(quote `*`atom`*`)` desugars to *`atom`*. For example:

- `(quote 251)` desugars to `251`
- `(quote #t)` desugars to `#t`
- `(quote "Hi there!")` desugars to `"Hi there!”`

Example:
`'(5 #f "cat" dog)` desugars to `(list 5 #f "cat" 'dog)`

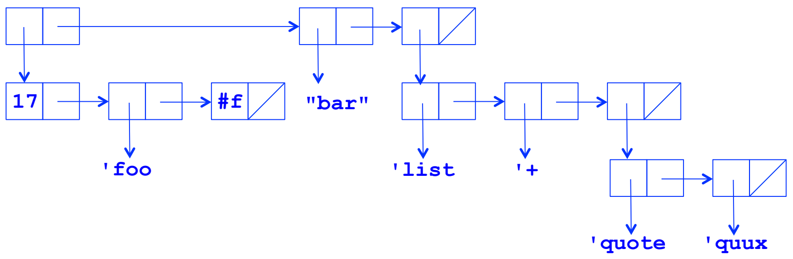
Quotation Exercise Solutions

1. Give the desugaring of the following quoted expression

```
'((17 foo #f) "bar" (list + (quote quux)))
```

```
(list (list 17 'foo #f)
      "bar"
      (list 'list '+ (list 'quote 'quux)))
```

2. Draw the box-and-pointer list structure of the value of this expression.



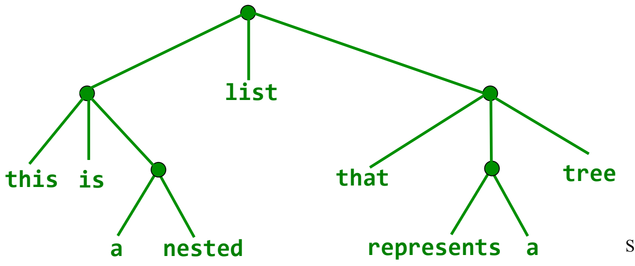
S-Expressions

Lisp pioneered ***symbolic expressions***, a.k.a. ***s-expressions***, a parenthesized notation for representing trees as nested lists (compare to other tree notations, like XML or JSON).

In these trees, nodes can have any number of children. Such trees are called ***rose trees*** ("rhododendron", in Greek).

An s-expression is just a quoted structure that represents a tree of intermediate nodes (lists) with leaves that are atoms.

Example: `'((this is (a nested)) list (that (represents a) tree))`

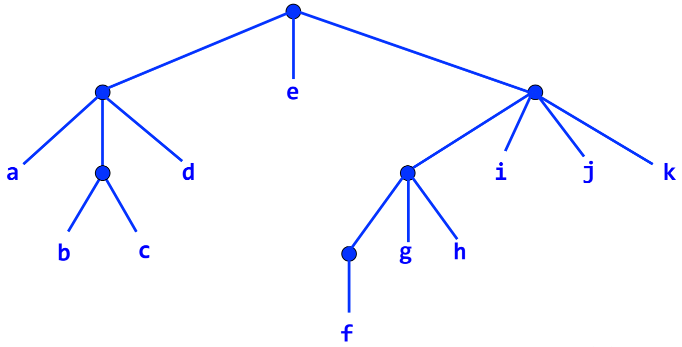


A sample s-expression Solutions

We will do some exercises with this sample s-expression:

```
(define tr '((a (b c) d) e (((f) g h) i j k)))
```

Draw the **tree** (not list structure) associated with this s-expression.



Functions on s-expression trees Solutions

Define the following functions that take an s-expression tree as their only arg:

1. `(sexp-num-atoms` *`sexp`*`)` returns the number of atoms (leaves) in the s-expression tree *sexp*

```
> (sexp-num-atoms tr)
11
```

```
(define (sexp-num-atoms sexp)
  (if (atom? sexp)
      1
      (foldr + 0 (map sexp-num-atoms sexp))))
```

2. `(sexp-atoms` *`sexp`*`)` returns a list of the atoms (leaves) encountered in a left-to-right depth first search of the s-expression tree *sexp*.

```
> (sexp-atoms tr)
'(a b c d e f g h i j k)
```

```
(define (sexp-atoms sexp)
  (if (atom? sexp)
      (list sexp)
      (foldr append '() (map sexp-atoms sexp))))
```

3. `(sexp-height` *`sexp`*`)` returns the height of the s-expression tree *sexp*.

```
> (sexp-height tr)
4
```

```
(define (sexp-height sexp)
  (if (or (atom? sexp) (null? sexp))
      0
      (+ 1 (foldr max 0 (map sexp-height sexp)))))
```

An s-expression Read-Eval-Print Loop (REPL)

```
(define (sexp-repl)
  (begin (display "Please enter an s-expression:")
     (let {[(sexp (read)]} ; read prompts user for sexp
       (if (eq? sexp 'quit)
            'done
            (begin (display (list 'sexp-num-atoms:
                                 (sexp-num-atoms sexp)))
                   (newline)
                   (display (list 'sexp-atoms:
                                 (sexp-atoms sexp)))
                   (newline)
                   (display (list 'sexp-height:
                                 (sexp-height sexp)))
                   (newline)
                   (sexp-repl))))))
```

On to Metaprogramming

A ***metaprogram*** is a program that manipulates another program, such as an interpreter, compiler, type checker, assembler, etc.

Q: In a metaprogram, how could we represent a Racket definition like this?

```
(define avg (lambda (a b) (/ (+ a b) 2)))
```

A: By adding a single quote mark!

```
'(define avg (lambda (a b) (/ (+ a b) 2)))
```

Does this give you a new appreciation for Lisp and what Paul Graham said about it?

Metaprogramming Example 1 Solutions

it's your turn

Define an `is-valid-lambda` function that takes an sexp and returns #t iff it is a valid Racket lambda expression. Assume parameters *must* be a list of identifiers, and that there is a single body expression. (Racket is actually more flexible than this.)

```
> (is-valid-lambda?
    '(lambda (a b)
       (/ (+ a b) 2)))
#t

> (is-valid-lambda?
    '(lamdba (a b)
       (/ (+ a b) 2)))
#f

> (is-valid-lambda?
    '(lambda foo
        (/ (+ a b) 2)))
#f

> (is-valid-lambda?
     '(lambda (a b)
        a b)
#f
```

```
(define (is-valid-lambda? sexp)
  (and (list? sexp)
       (= (length sexp) 3)
       (eq? (first sexp) 'lambda)
       (list? (second sexp))
       (forall? symbol? (second sexp))))
```

Metaprogramming Example 2 Solutions

it's your turn

Define a `desugar-let` function that takes an sexp that is a valid Racket let expression and transforms it to the application of a lambda.

```
> (desugar-let '(let ((a (* 2 3))
                      (b (+ 4 5)))
                  (- (* 10 a) b)))
'((lambda (a b) (- (* 10 a) b)) (* 2 3) (+ 4 5))
```

```
;; Note liberal use of readable syntactic abstractions!
(define (desugar-let sexp)
  (make-call (make-lambda (let-binding-names sexp)
                          (let-body sexp))
             (let-binding-defns sexp)))

(define (let-binding-names sexp) (map first (second sexp))
(define (let-binding-defns sexp) (map second (second sexp))
(define let-body third)

(define (make-lambda params body) (list 'lambda params body))
(define (make-call funexp argexps) (cons funexp argexps))
```