# Lecture 6 — August 19, 2015

- Today:
  - The quote form
  - Symbols
  - More on dot representations
  - Tagged data
  - The interpreter project
  - Mutation and environments
- Readings:
  - Finish *SICP* 2.5, 3.1, 3.2
  - Read *SICP* 3.3, 4.1, 4.4

**CANONICAL adj.** The usual or standard state or manner of something. A true story: One Bob Sjoberg, new at the MIT AI Lab, expressed some annoyance at the use of jargon. Over his loud objections, we made a point of using jargon as much as possible in his presence, and eventually it began to sink in. Finally, in one conversation, he used the word "canonical" in jargon-like fashion without thinking.

Steele: "Aha! We've finally got you talking jargon too!"
Stallman: "What did he say?"
Steele: "He just used 'canonical' in the canonical way."

*The Hacker's Dictionary*, edited by Guy Steele

CSE130, Summer Session II

The **`quote`** form prevents the interpreter from evaluating an expression:

```
> (+ 1 pi)
4.141592653589793
> (quote (+ 1 pi))
'(+ 1 pi)
```

Writing **`'datum`** is sugar for **`(quote datum)`**

Any ***Datum*** can be quoted:

*Datum* → *LexemeDatum* | *CompoundDatum*

*LexemeDatum* → *Boolean* | *Number* | *Character* | *String* | *Symbol*

*Symbol* → *Identifier*

*CompoundDatum* → **(** { *Datum* } **)**

*CompoundDatum* → **(** [ *Datum* ] **.** *Datum* **)**

*CompoundDatum* → *Abbreviation*

*Abbreviation* → **'** *Datum*

# Symbols

- Numbers, strings, booleans and characters have the same value when quote. But what's the type of the "value" of a quoted identifier?

```
> (define x 15)
> (number? x)
#t
> (quote x)
'x
> (number? (quote x))
#f
```

- It's a special type, *the **symbol** type:*

```
> (symbol? (quote x))
#t
```

# Symbols versus strings

- In MIT Scheme, strings are case sensitive, symbols are not
- In Racket, case matters for both:

```
(eq? (quote Apple) (quote APPLE)) ⟶ #f
(eq? (quote apple) (quote apple)) ⟶ #t
(string=? "Apple" "APPLE")        ⟶ #f
(string=? "Apple" "Apple")        ⟶ #t
```

- Strings are mutable, symbols are not
- Symbols are unique, and the unique identity can be tested using **`eq?`**.
  - Recall: In Java, don't use `==` on **`String`**s, unless you're sure they are the result of **`intern()`**
  - (*from the docs:* "Returns a canonical representation for the string object")
- Symbols are reminiscent of enums in C

# Quasiquoting

- Quoting is a nice way to make datums, but sometimes you don't want it to be so literal.
  - **`(quasiquote datum)`** — quote most of contents of a datum literally, except for what is ***unquoted***
  - **`(unquote datum)`** — unquote contents: contents will be evaluated and then replaced; works only in **`quasiquote`**.

```
> (define x 15)
> (quote (x x))
'(x x)
> (quasiquote (x (unquote x)))
'(x 15)
```

# Quasiquoting sugar

- `` `datum `` is sugar for `(quasiquote datum)`
- `,datum` is sugar for `(unquote datum)`

```
> `(+ 2 3)
'(+ 2 3)
> `(+ 2 ,(* 3 4))
'(+ 2 12)
> (let ([a 1] [b 2])
    `(,a . ,b))
'(1 . 2)
> '(`a)
'(`a)
> ''`a
''`a
> (let ([cp 'wocka])
    `(quasibear (,cp ,cp ,cp)))
'(quasibear (wocka wocka wocka))
```

# quasiquote unquote splicing, because reasons

- **`(unquote-splicing datum)`** — unquote contents: contents will be evaluated and then its elements are replaced in the surrounding list; works only in **`quasiquote`**

```
> (define nums (list 4 5 6))
> '(+ 1 2 3 ,nums)
'(+ 1 2 3 ,nums)
> `(+ 1 2 3 ,nums)
'(+ 1 2 3 (4 5 6))
> `(+ 1 2 3 (unquote-splicing nums))
'(+ 1 2 3 4 5 6)
> `(+ 1 2 3 ,@nums)
'(+ 1 2 3 4 5 6)
```

# The dotted-pair notation…

- The dotted-pair notation is also used by the interpreter when displaying cons cells where the **`cdr`** isn't a list (i.e., it isn't **`pair?`** or **`null?`**):

```
> (define x (cons 1 2))
'(1 . 2)
> (define y (cons 3 4))
'(3 . 4)
> (define z (cons x y))
'((1 . 2) 3 . 4)
```

- Note how `z` isn't `((1 . 2) . (3 . 4))`: because its **`cdr`** (which is `y`) *is* a pair!

# …is not a Scheme expression

- The dotted-pair notation is for printed representations of pairs and is not a Scheme expression whose value is a pair.
- For example, the following doesn't work:

```
> (define x (1 . 2))
. application: bad syntax in: (1 . 2)
```

- But the dotted-pair notation *can* be used in quotations:

```
> '(1 . 2)
'(1 . 2)
```

# Printing list structures

- Even in quotes there are syntactic restrictions:

```
> (define nums '(1 2 . 3 4))
. read: illegal use of ‘.’
> (define nums '(. 3 4))
. read: illegal use of ‘.’
```

```
(define (print-list-structure x)
  (define (print-pair x)
    (print-list-structure (car x))
    (cond [(null? (cdr x)) #f]
          [(pair? (cdr x)) (display " ")
                           (print-pair (cdr x))]
          [else (display " . ")
                (print-list-structure (cdr x))]))
  (cond [(null? x) (display "()")]
        [(pair? x) (display "(")
                   (print-pair x)
                   (display ")")]
        [else (display x)]))
```

# Tagged data

Idiom for creating new data types: Use a **`cons`** cell whose **`car`** describes the type and **`cdr`** describes the value.

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))

(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "expected a tagged item")))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "expected a tagged item")))
```

```
> (define bart (attach-tag 'person '(bart simpson)))
> (type-tag bart)
'person
> (contents bart)
'(bart simpson)
```

# Tagged data

`(cons 1 2)`

- Could be rational number $1/2$
- Could be complex number $1 + 2i$
- Could be width and height of an image

So, to prevent these “puns” we can be more specific:

```
> (attach-tag 'rational (cons 1 2))
'(rational 1 . 2)
> (attach-tag 'complex (cons 1 2))
'(complex 1 . 2)
> (attach-tag 'dimensions (cons 1 2))
'(dimensions 1 . 2)
```

# Tagged data

We don’t have to let the world know what tag or tags we’ve chosen for data items, so we can hide this information behind an abstraction:

```
(define (make-rational num den)
  (attach-tag 'rational (cons num den)))

(define (rational? x)
  (eq? (type-tag x) 'rational))

(define (make-complex real imag)
  (attach-tag 'complex (cons real imag)))

(define (complex? x)
  (eq? (type-tag x) 'complex))
```

Tagging allows for abstraction, but it isn’t encapsulation. *Why?*

# TokenScanner's nextToken method

```
import static org.instructures.interp.LexicalUtils.*;
/* ... */
public Lexeme nextToken() throws IOException {
  String tr = readNext(1);
  while (isWhitespace(tr) || tr.equals(";")) {
    if (tr.equals(";")) {
      reader.readLine();
    }
    tr = readNext(1);
  }
  if (tr.isEmpty()) {
    return new Lexeme(TokenType.EOF, "", reader.getLineNumber());
  } else if (tr.equals(".")) {
/* Many more cases... */
  } else if (isPunctuation(tr)) {
    return new Lexeme(parsePunctuation(tr), tr, reader.getLineNumber());
  }
  String message = String.format("Unexpected character: \"%s\"", tr);
  return new Lexeme(TokenType.ERROR,tr,message,reader.getLineNumber());
}

/* Returns a string containing as many of the given count of characters as possible. */
private String readNext(int nextCharsCount) throws IOException {
  char[] buff = new char[nextCharsCount];
  int actualLength = reader.read(buff, 0, buff.length);
  return (actualLength >= 1) ? new String(buff, 0, actualLength) : "";
}
```

# Helper methods in **`LexicalUtils`**

```
private static final HashMap<String, TokenType>
                          PUNCTUATION = new HashMap<>();
static {
  PUNCTUATION.put("(", TokenType.LPAREN);
  PUNCTUATION.put(")", TokenType.RPAREN);
  PUNCTUATION.put("[", TokenType.LBRACK);
  PUNCTUATION.put("]", TokenType.RBRACK);
  PUNCTUATION.put("'", TokenType.SQUOTE);
}

public static boolean isPunctuation(String tr) {
  return PUNCTUATION.containsKey(tr);
}

public static TokenType parsePunctuation(String tr) {
  return PUNCTUATION.get(tr);
}
```

# Use **peek** to look ahead

```
    } else if (tr.equals(".")) {
      String lookahead = peekNext(2);
      if (lookahead.equals("..")) {
        match(lookahead);
        return result(TokenType.SYMBOL, "...");
      }
      return result(TokenType.DOT, ".");
    } else if (tr.equals("#")) {
```

# **`read`** and **`write`**

- The output of the interpreter is equivalent to the output from the **`write`** procedure
  - **`(write obj)`** — prints the datum on the output
  - **`(read)`** — returns the next datum from the input
  - **`(eof-object? obj)`** — returns true if **`obj`** represents an EOF object
- This datum notation has been compared to XML (although it came well before XML)
- An ***S-Expression*** is what is written in this form
  - https://en.wikipedia.org/wiki/S-expression
  - It's closer to the now commonly-used JSON format

# `read` example

```
(define (get-next-number)
  (let ([datum (read)])
    (if (eof-object? datum)
        'done
        (let ([value (contents datum)])
          (cond
            [(rational? datum)
             (make-rational (car value) (cdr value))]
            [(complex? datum)
             (make-complex (car value) (cdr value))])))))
```

- DrRacket will prompt you for read input:

# Mutation!

- So far, we haven't once needed to use the assignment operator.
  - Pure functional programming doesn't even support assignment (e.g., the Haskell language).
- The `set!` special form:

```
> (define x 10)
> x
10
> (set! x "tofu")
> x
"tofu"
```

Why would this need to be a special form?

# The **`set!`** special form

- Syntax:
  - **`(set! identifier expression)`**
- Semantics:
  - Evaluate the second argument
  - Find the binding for the identifier symbol and change it to take on the new value
  - Value of the **`(set! .. ..)`** itself is unspecified (similar to **`define`**)

- By convention, predicates, which return Booleans, end with '?', similarly, mutators end with '!'

# No longer pure-functional

- Previously, procedures acted like mathematical functions: mapping inputs to outputs
- Substitution model:

```
> (define x 10)
> (+ x 5)
15
> (+ x 5)
15
```

expression has same value each time it is evaluated (in same scope as binding)

- Now, with assignment:

```
> (+ x 5)
15
> (set! x 94)
> (+ x 5)
99
```

expression's value depends on *when* it is evaluated

# Mutable form of cons cells

- Regular procedures:
  - **`(mcons a b)`** — create a ***mutable*** cons cell
  - **`(set-mcar! pair obj)`** — change the car of pair to obj
  - **`(set-mcdr! pair obj)`** — change the crd of pair to obj

```
> (define mutable-cell (mcons 1 1))

> mutable-cell
(mcons 1 1)

> (set-mcar! mutable-cell 2)

> mutable-cell
(mcons 2 1)

> (set-mcdr! mutable-cell mutable-cell)

> mutable-cell
#0=(mcons 2 #0#)
```

# Benefits of Assignment

- Similar to *SICP* 3.1.2

```
(define (estimate-pi trials)
  (set-rand-seed! 5)
  (sqrt (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))

(define (monte-carlo trials experiment)
  (let iter ((remaining trials)
             (passed 0))
    (cond ((= remaining 0) (/ passed trials))
          ((experiment)
           (iter (- remaining 1) (+ passed 1)))
          (else
           (iter (- remaining 1) passed)))))
```

# Estimate-pi as Purely Functional

```
(define (estimate-pi trials)
  (sqrt (/ 6 (monte-carlo trials cesaro-test 5))))

(define (cesaro-test r)
  (let* ((r1 (rand-update r))
         (r2 (rand-update r1)))
    (cons (= (gcd r1 r2) 1) r2)))

(define (monte-carlo trials experiment seed)
  (let iter ((remaining trials)
             (passed 0)
             (r seed))
    (cond ((= remaining 0) (/ passed trials))
          (else
           (let* ((result (experiment r))
                  (test-result (car result))
                  (new-r (cdr result)))
             (if test-result
                 (iter (- remaining 1) (+ passed 1) new-r)
                 (iter (- remaining 1) passed new-r)))))))
```