ECE570 Lecture 4: Interpretation

Jeffrey Mark Siskind

School of Electrical and Computer Engineering

Fall 2013

Symbols

SCHEME has a type *symbol* that is not common in other programming languages except PROLOG and other dialects of LISP.

Examples: `a`, `b`, `xyz`, `true`, `good`, `john`, `+`, `<=`

Symbols are written an sequences of characters without quotes.

The same set of characters are allowed as in variable names.

Lists can contain symbols. (Nested) lists containing symbols are often called *symbolic expressions* or *S-expressions*.

Examples: `(a b c)`, `((x y) (a b))`

S-expressions

S-expressions are often used to represent parsed sentences:

Examples: `(john ((saw mary) (with (a telescope))))`

or equations:

Examples: `(= (+ (sqr x) (sqr y)) (sqr z))`

or Boolean expressions:

Examples: `(implies (and p q) (or p r))`

or grammar rules:

Examples: `(s --> np vp)`

or mathematical assertions:

Examples: `(every x y z (= (+ (+ x y) z) (+ x (+ y z))))`

Quoting—I

In C, we need to differentiate between the variable `foo` and the string constant `foo`.

In a C program, we write the variable `foo` as `foo` and the string constant `foo` as `"foo"`.

This is called *quoting*.

The quotes are not part of the string.

`strlen("hello")` $\Longrightarrow$ `5`

`printf("hello")` $\Longrightarrow$ `hello`

Quoting—II

Likewise, in SCHEME, we need to differentiate between the variable `foo` and the constant symbol `foo`.

In a SCHEME program, we write the variable `foo` as `foo` and the symbol constant `foo` as `'foo`.

Note that the *syntax* of quoting symbols in SCHEME is a little different than quoting strings in C. There is an opening quote but no closing quote.

Like C, the quote is not part of the symbol.

`(list 'a 'b)` $\Longrightarrow$ `(a b)`

`(cons 'a (list 'a 'b))` $\Longrightarrow$ `(a a b)`

Quoting—III

In SCHEME, we can quote lists as well as symbols:

`'(a b c)` $\Longrightarrow$ `(a b c)`

`(append '(a b) '(c d))` $\Longrightarrow$ `(a b c d)`

`'(a b c)` is (almost) equivalent to `(list 'a 'b 'c)`.

Just like `x` is a variable access expression and `'x` is a constant expression,
`(+ 2 3)` is a procedure call expression and `'(+ 2 3)` is a constant expression.

`(list (+ 2 3) '(+ 2 3))` $\Longrightarrow$ `(5 (+ 2 3))`

Quoting—IV

Just as C programs are *represented* as *strings*, SCHEME programs are represented as *S-expressions* (i.e. lists).

The difference is that strings are flat. They must be parsed to determine their hierarchal structure.

Lists have inherent hierarchal structure and do not need to be parsed.

This makes it easier to write SCHEME programs to manipulate other SCHEME programs than to write C programs to manipulate other C programs.

Symbol Primitives

Symbols are a very impoverished type. The only things you can do with symbols is determine whether something is a symbol and whether two symbols are the same.

`(symbol? 'a)` $\Longrightarrow$ `#t`

`(symbol? 1)` $\Longrightarrow$ `#f`

`(eq? 'a 'a)` $\Longrightarrow$ `#t`

`(eq? 'a 'b)` $\Longrightarrow$ `#f`

`(eq? 'a 1)` $\Longrightarrow$ `#f`

Equality—I

Booleans:	`eq?`
numbers:	`=`
symbols:	`eq?`
characters:	`char=?`
strings:	`string=?`
procedures:	undecidable
lists:	???

Equality—II

```
(define (eqv? x y)
 (or (and (boolean? x) (boolean? y)
          (eq? x y))
     (and (number? x) (number? y)
          (= x y))
     (and (symbol? x) (symbol? y)
          (eq? x y))
     (and (character? x) (character? y)
          (char=? x y))
     (and (string? x) (string? y)
          (string=? x y))
     (and (null? x) (null? y))))
```

- The above is a white lie.

Equality—III

```
(define (equal? x y)
 (or (eqv? x y)
     (and (list? x)
          (list? y)
          (not (null? x))
          (not (null? y))
          (equal? (first x) (first y))
          (equal? (rest x) (rest y)))))
```

- The above is a white lie.

LET Syntax

$$\left.\begin{array}{l}\texttt{(let ((}v_1\ e_1\texttt{)}\\ \qquad\quad\texttt{(}v_2\ e_2\texttt{)}\\ \qquad\quad\vdots\\ \qquad\quad\texttt{(}v_n\ e_n\texttt{))}\\ \ e\texttt{)}\end{array}\right\} \rightsquigarrow \left\{\begin{array}{l}\texttt{((lambda (}v_1\ v_2\ \ldots\ v_n\texttt{)}\ e\texttt{)}\\ \ e_1\ e_2\ \ldots\ e_n\texttt{)}\end{array}\right.$$

LET* Syntax

$$\left.\begin{array}{l}\texttt{(let* ((}v_1\ e_1\texttt{)}\\ \qquad\quad\ \texttt{(}v_2\ e_2\texttt{)}\\ \qquad\quad\ \ \vdots\\ \qquad\quad\ \texttt{(}v_n\ e_n\texttt{))}\\ \ e\texttt{)}\end{array}\right\} \rightsquigarrow \left\{\begin{array}{l}\texttt{(let ((}v_1\ e_1\texttt{))}\\ \ \texttt{(let ((}v_2\ e_2\texttt{))}\\ \quad\vdots\\ \quad\texttt{(let ((}v_n\ e_n\texttt{))}\\ \quad\ e\texttt{)...))}\end{array}\right.$$

AND Syntax

$$
\left.\texttt{(and)}\right\} \rightsquigarrow \left\{ \texttt{\#t} \right.
$$

$$
\left.\texttt{(and }e\texttt{)}\right\} \rightsquigarrow \left\{ e \right.
$$

$$
\left.\texttt{(and }e_1\ e_2\ \ldots\ e_n\texttt{)}\right\} \rightsquigarrow \left\{ \begin{array}{l} \texttt{(if }e_1 \\ \qquad \texttt{(and }e_2\ \ldots\ e_n\texttt{)} \\ \qquad \texttt{\#f)} \end{array} \right.
$$

OR Syntax

$$
\begin{array}{r}
\texttt{(or)}
\end{array}
\Big\}
\rightsquigarrow
\Big\{\ \texttt{\#f}
$$

$$
\texttt{(or}\ e\texttt{)}\ \Big\}
\rightsquigarrow
\Big\{\ e
$$

$$
\texttt{(or}\ e_1\ e_2\ \ldots\ e_n\texttt{)}\ \Big\}
\rightsquigarrow
\left\{
\begin{array}{l}
\texttt{(let ((}v\ e_1\texttt{))} \\
\quad\texttt{(if}\ v \\
\qquad\qquad v \\
\qquad\qquad \texttt{(or}\ e_2\ \ldots\ e_n\texttt{)))}
\end{array}
\right.
$$

CASE Syntax

$$
\left.\begin{array}{l}
\texttt{(case}\ e\\
\ \texttt{((}s_1\texttt{)}\ \ e_1\texttt{)}\\
\ \texttt{((}s_2\texttt{)}\ \ e_2\texttt{)}\\
\ \vdots\\
\ \texttt{((}s_n\texttt{)}\ \ e_n\texttt{)}\\
\ \texttt{(else}\ \ e'\texttt{))}
\end{array}\right\}
\leadsto
\left\{\begin{array}{ll}
\multicolumn{2}{l}{\texttt{(let}\ \ \texttt{((}v\ \ e\texttt{))}}\\
\ \texttt{(cond} & \texttt{((eqv?}\ \ v\ \ s_1\texttt{)}\ \ e_1\texttt{)}\\
 & \texttt{((eqv?}\ \ v\ \ s_2\texttt{)}\ \ e_2\texttt{)}\\
 & \vdots\\
 & \texttt{((eqv?}\ \ v\ \ s_n\texttt{)}\ \ e_n\texttt{)}\\
 & \texttt{(else}\ \ e'\texttt{)))}
\end{array}\right.
$$

Mapping

```
(define (map procedure list)
 (if (null? list)
     '()
     (cons (procedure (first list))
           (map procedure (rest list)))))
```

Reduction—I

```
(define (list-sum list)
 (if (null? list)
     0
     (+ (first list) (list-sum (rest list)))))

(define (list-product list)
 (if (null? list)
     1
     (* (first list) (list-sum (rest list)))))
```

Reduction—II

```
(define (reduce procedure list identity)
 (if (null? list)
     identity
     (procedure (first list)
                (reduce procedure (rest list) identity))))

(define (list-sum list) (reduce + list 0))

(define (list-product list) (reduce * list 1))
```

A Calculator—I

```
(define (calculate expression)
 (cond ((number? expression) expression)
       ((list? expression)
        (case (first expression)
         ((+) (calculate-+ (rest expression)))
         ((-) (calculate-- (rest expression)))
         ((*) (calculate-* (rest expression)))
         ((/) (calculate-/ (rest expression)))
         ((expt) (calculate-expt (rest expression)))
         ((sqrt) (calculate-sqrt (rest expression)))))))
```

A Calculator—II

```
(define (calculate-+ arguments)
 (reduce + (map calculate arguments) 0))

(define (calculate-- arguments)
 (if (= (length arguments) 1)
     (- (calculate (first arguments)))
     (- (calculate (first arguments))
        (reduce + (map calculate (rest arguments)) 0))))
```

A Calculator—III

```
(define (calculate-* arguments)
 (reduce * (map calculate arguments) 1))

(define (calculate-/ arguments)
 (if (= (length arguments) 1)
     (/ (calculate (first arguments)))
     (/ (calculate (first arguments))
        (reduce * (map calculate (rest arguments)) 1))))
```

A Calculator—IV

```
(define (calculate-expt arguments)
 (expt (calculate (first arguments))
       (calculate (second arguments))))

(define (calculate-sqrt arguments)
 (sqrt (calculate (first arguments))))
```

Syntax vs. Procedure Calls—I

- `(+ 1 2)` is a *procedure call* expression.
- `(if p q r)` is **syntax**
  It is a *conditional* expression.
- `(lambda (x) (+ x 1))` is **syntax**.
  It is a *lambda* expression.
- `1` is a *constant* expression.
- `(quote a)` is **syntax**
  It is a *constant* expression (equivalent to `'a`).

Syntax vs. Procedure Calls—II

- `(define (f x) (+ x 1))` is **syntax**.
  It is a *definition*. (It isn't even an expression.)
- + is a *variable* that is *bound* to the addition procedure.
- `if`, `lambda`, and `define` aren't variables.
  (And they aren't bound to anything, procedures or otherwise.)
- `cond`, `let`, `let*`, `and`, `or`, and `case` are also **syntax**.

An Evaluator for micro-Scheme—I

```
(define (evaluate expression definitions bindings)
 (cond
  ((symbol? expression) (lookup-variable expression bindings))
  ((list? expression)
   (case (first expression)
    ((+) (evaluate-+ (rest expression) definitions bindings))
    ((-) (evaluate-- (rest expression) definitions bindings))
    ((*) (evaluate-* (rest expression) definitions bindings))
    ((/) (evaluate-/ (rest expression) definitions bindings))
    ((expt)
     (evaluate-expt (rest expression) definitions bindings))
    ((sqrt)
     (evaluate-sqrt (rest expression) definitions bindings))
    ((if) (evaluate-if (rest expression) definitions bindings))
    (else (evaluate-call expression definitions bindings))))
  (else expression)))
```

An Evaluator for micro-Scheme—II

```
(define (evaluate-+ arguments definitions bindings)
 (reduce +
         (map (lambda (expression)
               (evaluate expression definitions bindings))
              arguments)
         0))

(define (evaluate-- arguments definitions bindings)
 (if (= (length arguments) 1)
     (- (evaluate (first arguments) definitions bindings))
     (- (evaluate (first arguments) definitions bindings)
        (reduce +
                (map (lambda (expression)
                      (evaluate expression definitions bindings))
                     (rest arguments))
                0))))
```

An Evaluator for micro-Scheme—III

```
(define (evaluate-* arguments definitions bindings)
 (reduce *
         (map (lambda (expression)
               (evaluate expression definitions bindings))
              arguments)
         1))

(define (evaluate-/ arguments definitions bindings)
 (if (= (length arguments) 1)
     (/ (evaluate (first arguments) definitions bindings))
     (/ (evaluate (first arguments) definitions bindings)
        (reduce *
                (map (lambda (expression)
                      (evaluate expression definitions bindings))
                     (rest arguments))
                1))))
```

An Evaluator for micro-Scheme—IV

```
(define (evaluate-expt arguments definitions bindings)
 (expt (evaluate (first arguments) definitions bindings)
       (evaluate (second arguments) definitions bindings)))

(define (evaluate-sqrt arguments definitions bindings)
 (sqrt (evaluate (first arguments) definitions bindings)))
```

An Evaluator for micro-Scheme—V

```
(define (evaluate-if arguments definitions bindings)
 (if (evaluate (first arguments) definitions bindings)
     (evaluate (second arguments) definitions bindings)
     (evaluate (third arguments) definitions bindings)))
```

An Evaluator for micro-Scheme—VI

```
(define (lookup-variable variable bindings)
 (if (eq? variable (first (first bindings)))
     (second (first bindings))
     (lookup-variable variable (rest bindings))))

(define (lookup-definition name definitions)
 (if (eq? name (first (second (first definitions))))
     (first definitions)
     (lookup-definition name (rest definitions))))
```

An Evaluator for micro-Scheme—VII

```
(define (evaluate-call expression definitions bindings)
 (let ((definition
        (lookup-definition (first expression) definitions)))
  (evaluate
   (third definition)
   definitions
   (append (map (lambda (parameter argument)
                 (list parameter
                       (evaluate argument definitions bindings)))
                (rest (second definition))
                (rest expression))
           bindings))))
```