Scheme Expressions

- Scheme works by evaluating *expressions*
- Most Scheme systems use an interactive *read-eval-print-loop* (abbrev *REPL*):
  - The *read*: You enter an expression
  - The *eval*: The system determines the value of that expression
  - The *print*: The value of the expression is displayed
- An *expression* is simply an S-expression presented to the Scheme system for *evaluation*

Scheme Expressions (cont.)

- The value (or *meaning*) of a Scheme expression is defined case-wise

Non-Symbol Atoms

- The value of any non-symbol atom is that atom itself:

  `123` means `123`

  `123.456` means `123.456`

  `#t` means `#t`

  `#f` means `#f`

  `"a string"` means `"a string"`

- Non-symbol atoms are thus said to be *self-evaluating*

Quoted Expressions

- The value of the list `(quote` *x*`)` is *x*, for any S-expression *x*:

  `(quote 123)` means `123`

  `(quote #t)` means `#t`

  `(quote "a string")`

  means `"a string"`

  `(quote ())` means `()`

  `(quote (a b c))` means `(a b c)`

- `quote` is used to indicate *literals*

Quoted Expressions (cont.)

- In just about every implementation of Scheme, the expression

  `'`*x*

  is exactly the same as the expression

  `(quote` *x*`)`

Symbols

- The value of a symbol is the value that's currently *bound* to that symbol
- If a symbol is *unbound*, attempting to evaluate it as an expression is an error
- Scheme comes with a standard set of *top-level bindings*, primarily to provide a set of standard names for primitive operations
  - A symbol with a top-level binding is something like a top-level variable (but not exactly) in Python or C++

Special Forms

- The value of the list

  $(sym \;\; x_1 \;\; ... \;\; x_n)$

  if *sym* is any one of a number of specific, pre-determined symbols is defined by special rules
- We'll discuss these so-called *special forms* as we go

Applications

- The value of the list

  $(x_0 \;\; x_1 \;\; \ldots \;\; x_n)$

  when x_0 is not one of the special symbols indicating a special form is determined by *procedure application*
- Expressions x_0 ... x_n are evaluated, in an arbitrary order
- The value of x_0 must be a *procedure*
- The given procedure is then "called" with the values of x_1 ... x_n as the *actual parameters*

Procedures/Lambda-Expressions

- The value of the expression

  `(lambda (`sym_1 ... sym_n`) b)`

  is a *procedure*, having
  - Symbols sym_1 ... sym_n as its *formal parameters*
  - Expression *b* as its *body*
- The procedure may be applied to *n* actual parameters at a later point
- When applied, sym_1 is bound to the first actual parameter, sym_2 is bound to the second actual parameter, and so on

Procedures (cont.)

- After the formal parameters are bound, the body is evaluated
- Inside the body *b* of

  `(lambda (`sym_1 ... sym_n`) b)`

  sym_1 has the value of the first actual parameter, sym_2 has the value of the second actual parameter, and so on
- The value of the body *b* is returned as the value of the procedure application

Procedures (cont.)

- Procedures are just one more kind of atom in Scheme
  - Scheme procedures don't have the odd, fourth-class status that C++ functions have
- Scheme formal parameters are pretty much like C++ value formal parameters
- Scheme has pretty much the same scope rules as C++ (but Scheme had them first!)

Definitions

- The expression

  `(define` *sym x*`)`

  has a useless value, but produces an important *side-effect*:
  - It causes *sym* to be given a *top-level binding* (aka a *global binding*) that binds *sym* to the value of expression *x*
- `define`-expressions appear only as *top-level expressions*, never as *subexpressions* of larger expressions

Procedure Definitions

- The most common use of `define`'s is to give a symbol a *procedure binding*:

  `(define` *procedure-name*

  `(lambda (`*formal-name*$_1$ ... *formal-name*$_n$`)`

  body-expression`))`

- The symbol *procedure-name* can then be used to denote the procedure
- Inside *body-expression*, *formal-name*$_j$ is used to denote a formal parameter value

Example

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

Example

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A definition of symbol

```
3rd
```

to the value of the expression

```
(lambda (lis)
  (car (cdr (cdr lis))))
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A procedure (a `lambda`-expression) having formal parameter

```
lis
```

and body

```
(car (cdr (cdr lis)))
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

An application of the value of the expression

```
car
```

to the value of the expression

```
(cdr (cdr lis))
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A symbol, meaning the current value/binding for the symbol

```
car
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

An application of the value of the expression

```
cdr
```

to the value of the expression

```
(cdr lis)
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A symbol, meaning the current value/binding for the symbol

```
cdr
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

An application of the value of the expression

```
cdr
```

to the value of the expression

```
lis
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A symbol, meaning the current value/binding for the symbol

```
cdr
```

Example (cont.)

```
(define 3rd
  (lambda (lis)
    (car (cdr (cdr lis)))))
```

A symbol, meaning the current value/binding for the symbol

```
lis
```

Expression Summary

- Non-symbol atom *a*
  - Evaluates to *a*
- `(quote` *x*`)`, for any S-expression *x*
  - Evaluates to *x*
- Symbol *s*
  - Evaluates to the value currently bound to *s*
  - Produces an error if *s* has no binding

Expression Summary (cont.)

- Special form $(sym\ \ x_1\ \ ...\ \ x_n)$, for certain specific symbols *sym* and any S-expressions $x_1, \ldots x_n$
  - Special evaluation rules
- $(x_0\ \ x_1\ \ ...\ \ x_n)$, for any S-expressions $x_0, \ldots x_n$, where x_0 is not a symbol indicating a special form
  - Expressions x_0 ... x_n are evaluated, in an arbitrary order
  - Applies the value of x_0 to the values of $x_1, \ldots x_n$
  - Produces an error if value of x_0 isn't a procedure

Expression Summary (cont.)

- `(lambda (`sym_1 ... sym_n`) b)`, for any symbols sym_1 , … sym_n, and any S-expression *b*
  - Evaluates to a procedure, having symbols sym_1 ... sym_n as its formal parameters and expression *b* as its body
- `(define` *sym x*`)`, for any symbol *sym* and any S-expression *x*
  - Has a meaningless value, but gives *sym* a top-level binding that binds *sym* to the value of expression *x*