The Scheme Programming Language

References

The Scheme Programming Language by R. Kent Dybvig

`http://www.scheme.com/tspl2d/`

MIT Scheme Reference by Chris Hanson and others

`http://sicp.ai.mit.edu/Spring-2001/`
`manuals/scheme-7.5.5/doc/scheme_toc.html`

The Little Schemer by Daniel P. Friedman and Matthias Felleisen

Running scheme on onyx

MIT Scheme is installed on onyx. To run it, simply type

`scheme`

You can run scheme from inside vi by placing the following macro definition in your `.exrc` (or `.vimrc`) file

```
map @ :!scheme -load % ^M
```

The `^M` is entered as ctl-v ctl-m. This will automatically load the file I'm editing into the scheme interpreter when I press the @ key. Remember to save your file before running the macro.

To exit scheme, type `ctl-c q` or `(exit)`

Basic Elements

- atoms – literals and variables
  - numbers – just type them in
  - characters – `#\a` or `#\space`
  - strings of characters
  - boolean values – `#t` and `#f`
  - variables – a name (can't start with a digit) which is associated with a value of some kind
- expressions
- lists
- vectors
- pairs

numbers in Scheme

- Allowed Values: Scheme supports the usual numeric types like integer and real as well as rational and complex
- Operations:

`+ - * /`	usual arithmetic operations
number?	predicate to test for number
`=`	equality testing for numbers

- Literal Representation:

  `12 -2 25.4 6.7e-4`

Standard Procedures

Scheme provides a standard binding for a number of procedures

- arithmetic operators just use the usual operator as the name for the appropriate procedure : `+ - * /`
- similarly for comparison operators : `< > =`
- predicate functions : null? zero? number?

There is nothing to stop you from redefining them.

Booleans in Scheme

- Allowed Values: true and false
- Operations:

boolean?	check the type
eq?	equality test
not	change the truth value

- Literal Representation: #t and #f [sometimes represented as ()]

Characters in Scheme

- Allowed Values: the usual character set
- Operations:

char?	check the type
char=?	character equality
char->integer	get the ascii code
char-alphabetic?	is it a letter?
char-numeric?	
char-whitespace?	

- Literal Representation: `#\a` (for printable characters) `#\space` `#\newline` etc. for non-printable characters

strings in Scheme

- Allowed Values: sequences of characters
- Operations:

string-length	number of characters in string
string-append	add to a string
string->symbol	convert string to a symbol
string	convert list of characters to string
string-ref	get character at a particular position

- Literal Representation:

  characters surrounded by double quotes – `"a string"`

  argument to the quote procedure – `(quote quotation)`

symbols in Scheme

A symbol is just an identifier that is treated as a value.

For programs that manipulate other programs, it is useful to be able to treat an identifier as a special type.

- Allowed Values: any identifier
- Operations:

symbol?	type-check
eq?	equality test

- Literal Representation: `'name` or `(quote name)`

Variables

variable is a name that is associated with (bound to) a value

a variable *denotes* the value of its binding

variables are represented by identifiers

Identifiers

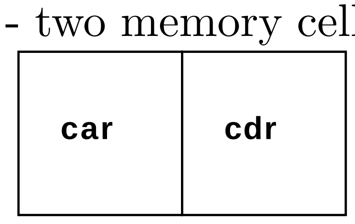
Scheme identifiers – letters, digits and many special characters

- **not** space or parentheses
- no digit at the beginning
- a few keywords are reserved – (define, if, cond)

Pairs

A pair is a compound data structure that serves as a building block for more complex data structures such as lists.

It is implemented as a cons cell - two memory cells which are the car and the cdr.



- Allowed Values:
- Operations:

pair?	test for pair (effectively also tests for lists)
eq?	test for same pair (similar to how == works for objects in Java)
cons	construct a pair
car	
cdr	

- Literal Representation: `'(a . c)`

lists in Scheme

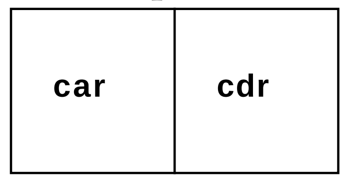
- Allowed Values: any ordered sequence of elements of arbitrary type
- Operations:

list	create a list
cons	add element to front of list
car	get first element
cdr	get all but first element
cadr, cddr, ...	combinations of car and cdr

- Literal Representation: `'(a b c) (quote a b c)`

Box and Pointer Diagrams

A pair, also called a cons cell is often represented using diagrams as shown below.

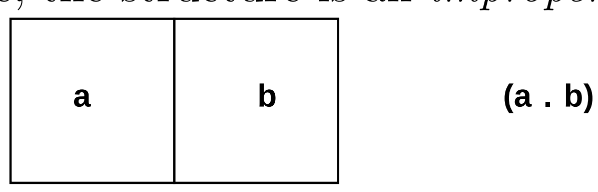


the car can be either a value or a pointer to another pair

the cdr can either be null or a pointer to another pair

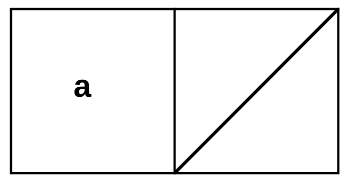
pairs may be shared by more than one list

if the cdr contains a value, the structure is an *improper list*

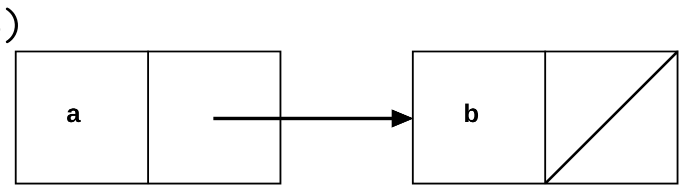


Example Box and Pointer Diagrams

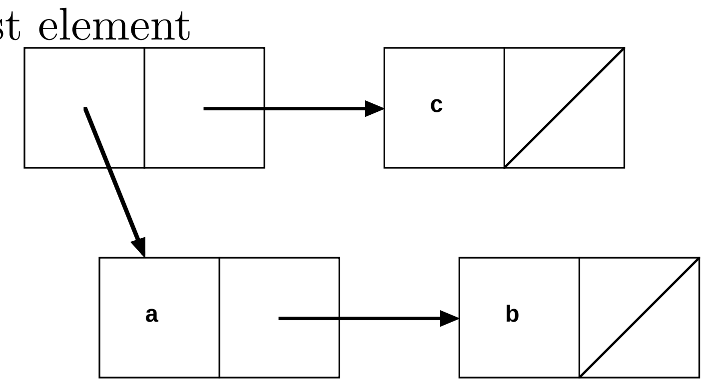
- one item list `( a)`



- two item list `( a b)`



- list with list as first element



Improper lists

If the cdr of a pair contains a value, the structure is an *improper list*

- not really a list at all
- use dot notation to represent improper lists `(a . d)`

vectors

- Allowed Values: any ordered sequence of values
- Operations:

vector	create a vector
vector?	test for a vector
vector-ref	get a particular element of a vector
vector-length	the number of elements in the vector
vector->list	convert vector to list
list->vector	convert list to vector

- Literal Representation: `#(a b)`

procedures

- Allowed Values:
- Operations:

procedure?	check for procedure
apply	use a list to supply the arguments to a procedure

- Literal Representation:

procedure call

Sequencing:

1. evaluate actual parameters
2. bind actual parameters to formal parameters
3. evaluate body

Special Forms in Scheme

- define
- if
- cond
- lambda
- and
- or

and

and and or are special forms to allow for short-cut evaluation

both can take any number of parameters (variatic like + and *)

```
(and test1 test2 ...testn)

(or test1 test2 ... testn)
```

define

```
(define variable expression)
```

Sequencing:

1. evaluate expression
2. bind value of expression to variable

$\boxed{\textbf{if}}$

```
(if test-expr then-expr else-expr)
```

Sequencing:

1. evaluate test-expr
2. if true, evaluate then-expr
3. otherwise, evaluate else-expr

cond

```
(cond
    (test1 conequent1)
    (test2 conequent2)
...
(else alternate))
```

Omitting the else results in an unspecified return value for an expression that executes that case.

This is equivalent to

```
(if test1 consequent1
           (if test2 consequent2
             ...
 alternative)...))
```

lambda

lambda is a special form for defining new procedures

`(lambda formals body)`

- formals is a list of formal parameters
- body is an expression (or list of expressions)

lambda doesn't name the procedure; use define for that

scheme allows anonymous procedures

types aren't needed because scheme keeps track of them

alternate form for function definitions

```
(define (name formals) body)
```

variable-arity procedures

Use a special form of lambda

(lambda formal body)

where formal is a single argument which must be a list

```
(define show
    (lambda items
        (if (not (null? items))
            (begin (display (car items)) (apply show (cdr items)))
            )))
```

Local Binding

We've seen two ways to create bindings

1. use define to create top-level bindings – region is the entire program
2. lambda expressions create local bindings for parameters – region is the body of the procedure

We often want to create local bindings for temporary use.

You can't do a define inside a lambda expression. You need one of two special forms to do this: let and letrec.

Special Form: let

```
(let ((var1 expr1)...(varn exprn))
     body)
```

This is syntactically equivalent to

```
((lambda (var1 ... varn) body)
    expr1 ...exprn)
```

This is an example of a special form that really isn't necessary. However, it is very nice to have. Such special forms are called *syntactic abstractions* and are often referred to as *syntactic sugar*.

let can also be used to define local functions. However it cannot be used to define a function that calle itself recursively.

Special Form: letrec

letrec allows you to define local functions that are recursive.

```
(letrec ((var1 expr1)..(varn exprn))
    body)
```

The region for the bindings includes all the expressions.

apply

Sometimes, you have a list and you want to use the elements of the list as the arguments for some procedure.

```
(apply proc lst)

(apply + (2 3 4))

(apply car '(a b c))
```

map

`(map proc lst)`

apply procedure proc to each element of lst – result is a new list with the same number of elements

```
(map (lambda (x) (* x x) '(1 2 3 4 5)))
```

Programs

a Scheme program is a sequence of definitions and expressions executed in order

Anything following a semicolon (`;`) is considered a comment

Once inside the interpreter, you can load a file containing scheme expressions

```
(load "defs.scm")
```

where defs.scm is the name of the file to be loaded.

You can also start up the interpreter and read stuff from a file by the command

```
scheme -load "defs.scm"
```