1 Initial Instructions

You should have read or at the minimum start reading the following.

- Chapter 1 and section 2.2 of Structure and Interpretation of Computer Programs,
- Chapters 1-2 of The Little Schemer,
- Scheme Official Manual – The Revised\textsuperscript{5} Report on Scheme, refer to it for details.

During this lecture, as a student you should note both the

- terminology & perspectives
- facts

Note that during these lecture, we will study Scheme from the perspective of languages in general, not just how to program in Scheme.

2 The Scheme Type Universe

Most programming languages split values up into different types. Scheme also splits values into several types as shown in Figure\textsuperscript{H}

![Figure 1: The Universe of Types in Scheme](image)

The type null contains just (). The domain of type null is a subset of the domain of type list. The domain of type list is almost a subset of the domain of type pair, except () is a list, and not a pair.

Each subset, except datum, has a predicate associated with it to test if the value is of that type. For example, for pair there is the predicate pair?, for list there is the predicate list?, etc.

Exercise 2.1 Are there any programming languages that do not split values into types?

Some examples are turing machines, BCPL, etc that do not split values into different types.
Exercise 2.2  What’s the advantage of splitting values into types?

Main advantage of splitting values into types is that the language figures out for you what piece of code to run based on the type information. For example, consider the operation \texttt{add}. This operation may be used to add numbers but there are variant of this operation for different kinds of numbers, e.g. int vs. float vs. double. Giving types frees programmers from specifying variants of operations thus simplifies their tasks.

Exercise 2.3  What operator in Java or C# is similar to these predicates, but a bit more general?

3  Characteristics of Data Types in Programming Languages

Exercise 3.4  What characterizes (specifies) an Abstract Data Type (ADT)?

values and operations

More concretely a data type or an abstract date type is specified by the following components.

- Values
  - abstract
  - external (printed form)
- Operations
  - procedures
  - syntax of literals
  - special forms

3.1 Characteristics of the boolean type

- Values
  - abstract : true, false
  - external (printed form) : #t, #f
- Operations
  - procedures : not, eqv?
  - syntax of literals : #t, #f
  - special forms : if, cond, and, or

We don’t usually distinguish the external (output) form from the syntax of literals (input). Finally, special forms could be defined by macros in programming.
3.2 Characteristics of the number type

See the Revised\textsuperscript{5} Report on Scheme (R5RS) for details about procedures on numbers, such as: +, −, *, /, \texttt{zero?}, \texttt{positive?}, \texttt{negative?}, \texttt{odd?}, \texttt{even?}, <, >, <=, >=, \texttt{abs}, \texttt{quotient}, \texttt{remainder}, \texttt{modulo}, \texttt{max}, \texttt{min}, \texttt{floor}, \texttt{ceiling}, \texttt{truncate}, \texttt{round}, \texttt{expt}, \texttt{log}.

Example 3.1 Write a procedure "average" that takes two numbers and returns their average.

\texttt{(define average (lambda (x y) (/ (+ x y) 2)))}

Exercise 3.5 Can you write a procedure "mod7" that returns the result of taking its argument modulo 7?

Some examples of the output of such procedure that we have called \texttt{mod7} are the following.

\texttt{(mod7 8) \rightarrow 1}
\texttt{(mod7 23) \rightarrow 2}

3.3 Characteristics of the string type

\begin{itemize}
\item Values
  \begin{itemize}
  \item abstract : finite sequences of chars
  \item external (printed form) : "a string", "hi"
  \end{itemize}
\item Operations
  \begin{itemize}
  \item procedures : \texttt{string}, \texttt{string–append}, \texttt{string–ref} (0 based), \texttt{string–length}, \texttt{string=?} \texttt{string <=?}, \texttt{substring}, etc.
  \item syntax of literals : "a string", literals almost exactly as in C or C++.
  \item special forms : ...
  \end{itemize}
\end{itemize}

Some examples of the usage of the procedure \texttt{substring} available for this type are shown below.

\texttt{(substring "a string" 0 3) \rightarrow "a s"}
\texttt{(substring "a string" 1 3) \rightarrow "s"}
\texttt{(substring "a string" 2 3) \rightarrow "s"}
\texttt{(substring "a string" 3 3) \rightarrow "s"}
\texttt{(substring "a string" 2 5) \rightarrow "str"}

3.4 Symbols

A symbol in Scheme is a string of characters starting with anything except parenthesis. For example, ‘\texttt{a–symbol}, ‘\texttt{hrides}, ‘\texttt{coms342} are symbols. The type symbol in Scheme is like strings, except for the following two properties.

\begin{itemize}
\item Symbols provide constant time equality test, and
\item can’t be mutated.
\end{itemize}

Exercise 3.6 What is not a symbol?

An example would be (‘\texttt{a–symbol}).
Exercise 3.7  What is it?

3.4.1 Characteristics of the symbol type

- Values
  - abstract: nonempty finite sequences of characters
  - external (printed form): a-symbol, mySymbol, +, ...

- Operations
  - procedures: eqv?, eq? (fast equality test)
  - syntax of literals: "a string", literals almost exactly as in C or C++.
  - special forms: quote, ’
    * Examples of using special forms.

    (quote a-symbol)
    ’a-symbol
    ’+
    ’if
    ’lambda
    ’quote

    * quotation even works for keywords, important as we’ll be using these as symbols in what
      we do.

3.5 Lists

The list data type in Scheme works like a singly linked lists. The elements are accessed at head (so also like
stacks). Some examples of lists are:

’()
’(1 2)
’(a b c)

These are from top-to-bottom, a list containing nothing, a list containing two symbols 1 and 2, a list
containing three symbols a, b, and c, respectively. Following are examples of Scheme expressions that also
produce the same lists but instead of the quotation use the constructor list.

> (list )
()  
> (list 1 2)
(1 2)  
> (list ’a ’b ’c)
(a b c)

Note that in the lecture notes, we use the notation > to denote the prompt of the Scheme interpreter and
the following line as output to demonstrate an interactive session.

Following is another example of a Scheme expression that produces same lists but instead use the oper-
ator cons.

> (cons 1 (cons 2 ’()))
(1 2)  
> (cons ’a (cons ’b (cons ’c ’())))
(a b c)

Exercise 3.8  Try examples of other operations on lists such as car, cdr, append, cadr, caddr, caar, etc.
3.5.1 Characteristics of the list type (More precisely list-of T)

- Values
  - abstract: finite sequences of T
  - external (printed form): (), (a b), (c d e), ...

- Operations
  - procedures: cons, car, cdr, null?, length, append, list, ...
  - syntax of literals: ’(), ’(a b), ...

3.5.2 Box and Pointer View vs. DOT Notation View

In understanding the results of evaluating list-related operations it often makes sense to express lists using two different notations namely box and pointer view and the dot notation view. The dot notation view is used in the book as a notation for list and pair values. Following illustrates these views.

<table>
<thead>
<tr>
<th>BOX AND POINTER VIEW</th>
<th>DOT NOTATION VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>( 1 . (2 . ()))</td>
</tr>
<tr>
<td>car</td>
<td>car</td>
</tr>
<tr>
<td>cdr</td>
<td>cdr</td>
</tr>
</tbody>
</table>

For example, the evaluation of the expression (cons 1 (cons 2 ’())) results in the following list (expressed using box and pointer views as well as using the dot notation view).

<table>
<thead>
<tr>
<th>BOX AND POINTER VIEW</th>
<th>DOT NOTATION VIEW</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram" /></td>
<td>(1 . (2 . ()))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>()</td>
</tr>
</tbody>
</table>

While drawing similar diagrams for lists in homework and exams, please note that the following is acceptable, but we prefer the one above.

| [ 1 | *-]--->[ 2 | / ]

Exercise 3.9 What’s the dot notation for the following?

- (cons 4 (cons 5 6))
- (cons ’a ’b)

Exercise 3.10 The Scheme interpreter prints with dots only when it has to, otherwise uses list notation. #t/#f?

Exercise 3.11 The expressions ’(a b c), (list ’a ’b ’c), and (cons ’a (cons ’b (cons ’c ’()))) all evaluate to the same value. Draw box and pointer diagrams for the result.
Exercise 3.12  Please provide the dot notation equivalents for the following expressions.

<table>
<thead>
<tr>
<th>expression</th>
<th>dotted</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>'()</td>
<td>()</td>
<td>()</td>
</tr>
<tr>
<td>(cons 1 '())</td>
<td>(1 . ()</td>
<td>(1)</td>
</tr>
<tr>
<td>(cons 1</td>
<td>(cons 2 '())</td>
<td></td>
</tr>
<tr>
<td>(cons 1 2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(cons (cons 1 '())</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.5.3 Pairs vs. Lists

Definition 3.2 A pair (or cons cell) is what is returned by cons.

For example, both (cons 1 2) and (cons 1 (cons 2 '())) are pairs.

Definition 3.3 A list is either () or a pair whose cdr is a list.

For example, both () and (cons 1 (cons 2 '())) are lists.

Definition 3.4 An improper list is a pair that is not a list.

For example, (cons 1 2) is a pair but not a list.

Exercise 3.13  How do you test whether a value is not a list?

Exercise 3.14  So is list a subset of pair?

No

Exercise 3.15  Is null a subset of list?

Yes
Exercise 3.16 \ Is an improper list a list?

No

3.5.4 Examples Using Proper Lists

We will now illustrate some examples in Scheme using proper lists.

Example 3.5 Write a procedure to return the second element of a list.

\[
\text{\texttt{(define second}} \\
\text{\texttt{(lambda (ls) \)}} \\
\text{\texttt{(cadr ls))}}
\]

Exercise 3.17 \ How would you test that?

Exercise 3.18 \ Can you write a procedure to return the third element of a list?

Example 3.6 Write \texttt{yodaize}, a procedure with the following behavior:

\[
>\text{\texttt{(yodaize '(you will study))}} \\
\text{\texttt{(study you will))}} \\
>\text{\texttt{(yodaize '(you must eat))}} \\
\text{\texttt{(eat you must))}}
\]

Two solutions are shown in Figure 2 and Figure 3. The solution in Figure 2 uses the list operation, whereas the solution in Figure 3 uses cons operation. These examples also demonstrate defining and calling functions in Scheme.

\[
\text{(define yodaize}} \\
\text{(lambda (ls) \)}} \\
\text{(list (third ls) (first ls) (second ls))})
\]
\[
\text{(define first (lambda (ls) (car ls))})
\]
\[
\text{(define second (lambda (ls) (cadr ls))})
\]
\[
\text{(define third (lambda (ls) (caddr ls))})
\]

Figure 2: An example Scheme program that yodaizes argument strings.

3.6 Characteristics of Vectors or More Precisely (vector-of T)

Vectors in Scheme allow random access (vs. lists, which are sequential).

- Values
  - abstract: finite sequences of cells containing T objects
  - external (printed form): #(), #(a b), #(c d e), ...
(define yodaize
(lambda (ls)
  (cons (third ls)
        (cons (first ls)
              (cons (second ls)
                   '())))))
(define first (lambda (ls) (car ls)))
(define second (lambda (ls) (cadr ls)))
(define third (lambda (ls) (caddr ls)))

Figure 3: Another implementation of the yodaize function.

- Operations
  - procedures: vector, make-vector, vector-ref, vector-set!, vector-length, ...
  - syntax of literals: '#()', '#(a b)....

Example 3.7 Some examples of using vectors in Scheme. Note that both methods of constructing a vector produce same result.

> '#(1 2 3)
#(1 2 3)
> (vector 1 (+ 1 1) (+ 2 1))
#(1 2 3)

Following are some important properties of vectors in Scheme and similarity/differences from similar concept in other languages.

- zero-based indexing (as in C/C++, Java, but not Pascal)
- may be heterogeneous (unlike C++, Pascal, like Smalltalk)
- carry their length (convenient, unlike C/C++, like Java, Smalltalk)
- checked bounds (unlike C/C++, like Pascal, Java, Smalltalk)

3.7 Equality Predicates

Here are some example usage of equality predicates. Please see Revised5 Report on Scheme, section on Equality predicates.

> (eq? (cons 1 '()) (cons 1 '()))
#f
> (equal? (cons 1 '()) (cons 1 '()))
#t
> (equal? (vector 1 2 3) (vector 1 2 3))
#t
> (eq? (vector 1 2 3) (vector 1 2 3))
#f
> (equal? 9876543210 9876543210)
#t
> (equal? 9876543210 9876543210)
#t

Exercise 3.19 Create examples that use eq? vs. equal? for lists, vectors, big numbers.
4 Syntactic Parts of a Programming Language

- program
  - definition
  - statements
  - expressions
  * literals (\#t, \#f, "a str")
  * variable references (x, ls)
  * procedure calls ((f x), (car ls))

Note: every parenthesis is important; you can’t leave any out, or put in any extra.

Exercise 4.20 What’s the translation of 5 + 6 + 7 into Scheme?

Exercise 4.21 How would sqrt(cos(x + 5)) be translated into Scheme?

Exercise 4.22 What’s the syntax of a procedure call in Scheme?

<call> ::= (<expression> <expression>*)

Exercise 4.23 What’s a rule for forming the Scheme from the algebraic notation?

This is an example of regularity (as mentioned in the designing-a-language unit).

5 Definitions

Below are Scheme syntax examples for definitions.

(\texttt{define pi}
 3.14159)

(\texttt{define greeting}
 "Welcome!"
)

This is an example of \textbf{special forms} in scheme.

- provide the "magic"
- don’t evaluate their all arguments in random order, as do procedures

Exercise 5.24 What is like this in C++? Pascal?
Another example of definitions follows.

```scheme
(define add1
  (lambda (n) (+ n 1)))

(define is-pi?
  (lambda (n) (= n pi)))
```

**Exercise 5.25** What is main difference between the definition of pi and definition of add1?

## 6 Programs

Scheme syntax: a series of definitions and expressions Following are some example Scheme programs.

```scheme
(define pi
  3.14159)
(define greeting
  "Welcome!")
(define is-pi?
  (lambda (n) (= n pi)))

(display greeting)
(newline)
(is-pi? pi)
```

The semantics is defined such that programs are executed from top (start of file) to end.

## 7 Read-Eval-Print Loop

The scheme interpreter runs a giant infinite loop that essentially follows a sequence of three steps as shown below. Most of the work in the read step involves parsing of input (also called front-ends in compiler designs), whereas eval step is where most of the language semantics is implemented.

```
(exit)
---> read ---->
^ \ \\
/ v
print eval
^-----/
```

## 8 If expressions

Here are an example transcript of running Scheme examples that use the if expression.

```scheme
> (if #t 3 4)
3
> (define x 2)
> (if (zero? x) (+ x 3) x)
2
> (if (< x 0) (- x) x)
2
```

**Exercise 8.26** So what’s the Scheme syntax?
Exercise 8.27  Is the else-part required? Why would it be?

Exercise 8.28  How does this differ from an if statement in C++ or Java?

Exercise 8.29  What’s this like in C/C++/Java/C#? Anything like this in Visual Basic? (like `e1 ? e2 : e3` in C/C++/Java/C#)