Scheme: Strings & I/O

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, April 12, 2019

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Review
Introduction to Scheme [1/2]

Scheme is a Lisp-family PL with a minimalist design philosophy.

Scheme code consists of parenthesized lists, which may contain atoms or other lists. List items are separated by space; blanks and newlines between list items are treated the same.

```
(define (hello-world)
    (begin
        (display "Hello, world!")
        (newline)
    )
)
```

When a list is evaluated, the first item should be a **procedure** (think “function”); the remaining items are its arguments.

Review
Introduction to Scheme [2/2]

The type system of Scheme is similar to that of Lua.

- Typing is dynamic.
- Typing is implicit. Type annotations are generally not used.
- Type checking is structural. Duck typing is used.
- There is a high level of type safety: operations on invalid types are not allowed, and implicit type conversions are rare.
- There is a fixed set of types (36 of them).

Scheme has 36 types (as compared to Lua's 8).

Review
Scheme: Data — Evaluation [1/2]

`quote` is a special procedure that takes one parameter, suppressing the normal parameter evaluation. It returns this parameter.

```
> (quote (1 2 3))
(1 2 3)
```

The leading-single-quote syntax is actually shorthand for `quote`.

```
> '(1 2 3)  ; Same as (quote (1 2 3))
(1 2 3)
```

For code from this topic, see `data.scm`*.*

Review
Scheme: Data — Evaluation [2/2]

`eval` is a procedure that takes one parameter and evaluates it. `eval` does not suppress the normal evaluation of parameters, so, strictly speaking, evaluation happens twice: the parameter is evaluated, and then it evaluates the result.

```
> (eval '(+ 2 3))
5
```

A variation is `apply`. This takes a procedure and a list of arguments. It calls the procedure with the given arguments and returns the result.

```
> (apply + '(2 3))
5
```

Scheme: Data — Data Format [1/5]

The dot notation originally used in S-expressions is also valid in Scheme.

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

A list is really shorthand for the equivalent dot notation, again, just as in the original S-expression syntax.

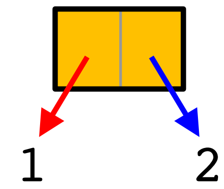
```
> '(1 . (2 . (3 . (4 . ()))))
(1 2 3 4)
```

Dot (`.`) is *not a procedure*. It is simply another way of typing S-expressions. If you want a procedure that puts things together the way dot does, use `cons`.

Scheme: Data — Data Format [2/5]

The main building block for constructing data structures in Scheme is the **pair**. You can think of this as a node with two pointers.

(1 . 2)



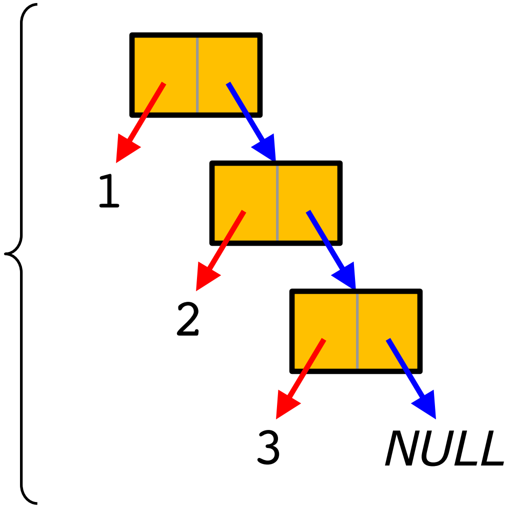
We get the item referenced by the left pointer using `car`; similarly use `cdr` for the right pointer.

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

Scheme: Data — Data Format [3/5]

Lists are constructed from pairs and null.

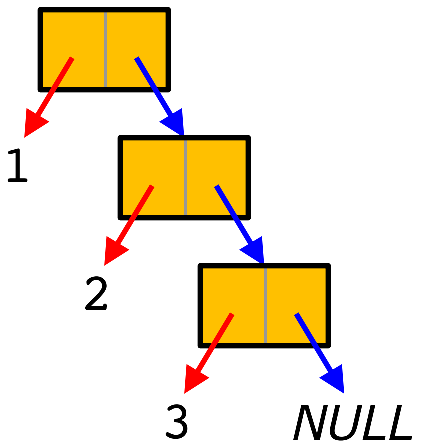
```
(1 2 3) = (1 . (2 . (3 . ())))
```



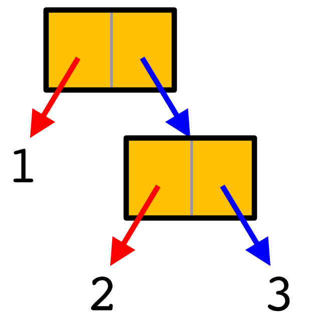
Scheme: Data — Data Format [4/5]

The full story on the dot syntax is that the dot may optionally be added just before the *last* item of a list.

```
(1 2 3)
= (1 . (2 . (3 . ())))
```



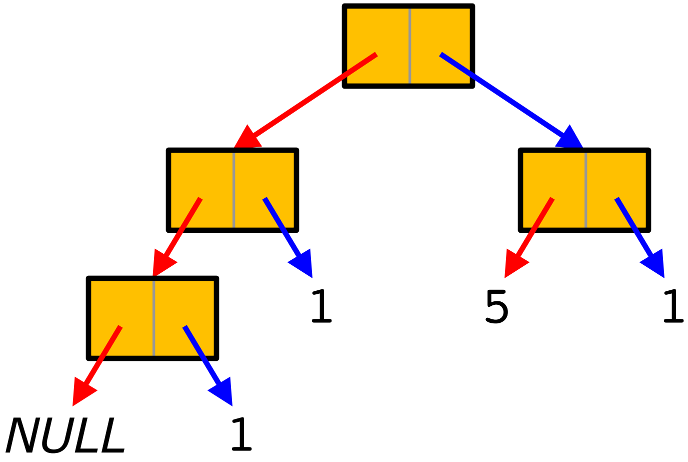

```
(1 2 . 3)
= (1 . (2 . 3))
```



Scheme: Data — Data Format [5/5]

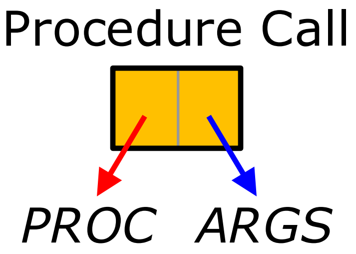
We can create arbitrary binary trees—with the restriction that only leaf nodes contain data.

```
(((((() . 8) . 1) . (5 . 1))
```



Scheme: Data — Varying Number of Parameters

A procedure call is a pair: `(`*PROC* `.` *ARGS*`)`.



`define` will take this form of a “picture” of a procedure call.

```
(define (sum . args)
  …
)
```

`args` is a list of the arguments of `sum`. So `sum` can take an arbitrary number of parameters.

How to make a recursive call on `(cdr args)`?

```
(sum . (cdr args))              ; WRONG!
(eval (cons sum (cdr args)))    ; Okay
(apply sum (cdr args))          ; Okay (and also clearer)
```

`(sum . (cdr args))` is just another way to write `(sum cdr args)`, which is not what we want.

Review
Scheme: Data — Lazy Evaluation [1/3]

Normal evaluation in Scheme is eager.

However, we can do lazy evaluation in Scheme, using a **promise**: a wrapper around an expression that leaves the expression unevaluated.

When we **force** a promise, the expression is evaluated, and the resulting value is stored in the promise and returned. Force again, and the same value is returned, without reevaluating the expression.

Scheme: Data — Lazy Evaluation [2/3]

Create a promise using `delay`.

```
> (define pp (delay (* 20 5)))
```

The type-checking predicate for promises is `promise?`.

```
> (promise? pp)
#t
```

Force a promise using `force`. Again, force a promise as many times as you like; evaluation only happens the first time.

```
> (force pp)
100
```

Review
Scheme: Data — Lazy Evaluation [3/3]

Using promises, we can create the kind of lazy infinite lists we saw in Haskell (rather less conveniently, though).

One way to do this is to construct a list as usual, from pairs and null, but wherever there is a pair, we actually have a promise wrapping a pair.

TO DO

- Write code to create a lazy infinite list.
- Write code to print out a portion of the above list. (With a little thought, we can write a procedure that will print both lazy lists and normal lists.)

See `data.scm`.

Scheme: Strings & I/O
Strings [1/3]

As in so many PLs, to understand Scheme I/O, it helps to know something about Scheme strings.

String literals in Scheme are surrounded by double quotes. The usual backslash escapes are accepted.

```
"This is a string."
"A newline: \nA double quote: \" A backslash: \\"
```

Check whether a value is a string with `string?`.

```
> (string? "42")
#t
> (string? 42)
#f
```

For code from this topic, see `string.scm`.

Strings [2/3]

Get the length of a string with `string-length`.

```
> (string-length "Hello!")
6
```

Concatenate strings with `string-append`.

```
> (string-append "abc" "def" "ghi" "jklmnop")
"abcdefghijklmnop"
```

Get a substring with `(substring` *STRING START PAST_END*`)`.

```
> (substring "Howdy thar!" 2 7)   ; Zero-based indices
"wdy t"
```

Includes the characters at indices 2, 3, 4, 5, 6, but *not* 7.

Strings [3/3]

Convert a number to a string using `number->string`.

```
> (number->string 42)
"42"
```

Convert a string to a number using `string->number`. This returns the number, or `#f` if the conversion could not be done. So the result can be used in an `if`.

```
> (string->number "42")
42
> (string->number "Hello!")
#f
```

`(if` *COND THEN-EXPR ELSE-EXPR*`)`

When the above is evaluated, *THEN-EXPR* is chosen if *COND* evaluates to *anything* other than `#f`.

Scheme: Strings & I/O
Characters [1/2]

Character literals generally have the form #*CHAR*. Some characters have special literals.

```
#\A   ; The 'A' character
#\newline  #\space
```

Check whether a value is a character with `char?`.

```
> (char? #\x)
#t
> (char? "x")
#f
> (string? #\x)   ; A Scheme character is not a string
#f
```

Characters [2/2]

Convert a character to its numeric version (ASCII value/Unicode **codepoint**) with `char->integer`. Reverse: `integer->char`.

```
> (char->integer #\A)
65
> (integer->char 65)
#\A
```

Convert between strings and lists of characters with `string->list` and `list->string`.

```
> (string->list "Howdy thar!")
(#\H #\o #\w #\d #\y #\space #\t #\h #\a #\r #\!)
> (list->string '(#\a #\b #\c))
"abc"
```

Scheme: Strings & I/O
Comparisons [1/5]

We have seen the Scheme numeric comparison operators: `=` `<` `<=` `>` `>=`. These can only be used with numbers.

Some Scheme types have their own comparison functions.

```
> (string=? "abc" "def")
#f
> (string=? "42" 42)
ERROR
> (string<? "abc" "def")
#t
```

Also: `string<=?` `string>?` `string>=?` `char=?` `char<?` `char<=?` `char>?` `char>=?`

Scheme: Strings & I/O
Comparisons [2/5]

There are several kinds of equality in Scheme.

The simplest is `eq?`, which means “same location in memory”.

```
> (eq? '() '())
#t
> (eq? 2 2)
IMPLEMENTATION-DEPENDENT
> (define a '(1 2))
> (eq? a '(1 2))
#f
> (define b a)
> (eq? a b)
#t
```

Scheme: Strings & I/O
Comparisons [3/5]

Next is `eqv?`, which means "same primitive value".

```
> (eqv? 2 2)
#t
> (eqv? 2 2.0)
#f
> (define a '(1 2))
> (eqv? a '(1 2))
#f
> (eqv? "ab" "ab")
```

IMPLEMENTATION-DEPENDENT

> Lists and strings are not primitive values.

Scheme: Strings & I/O
Comparisons [4/5]

Then there is `equal?`, which does the following:

- If the types are different, then return `#f`.
- For primitive values (everything we have covered except strings and pairs) of the same type, call `eqv?`.
- For strings, call `string=?`.
- For pairs, recursively call `equal?` on the `car`s & `cdr`s.

```
> (define a '(1 2))
> (equal? a '(1 2))
#t
> (equal? "ab" "ab")
#t
```

Scheme: Strings & I/O
Comparisons [5/5]

`equal?` mostly does what we usually want, with one caveat. Since it always returns `#f` when the types are different, it can give unexpected results with numbers.

```
> (equal? 2 2.0)
#f
```

I offer the following rule of thumb.

- Use = for numeric equality.
- Use `equal?` for most other kinds of equality.
- If you want the code to indicate what type you are comparing, and flag type errors for other types, then use a type-specific equality function (e.g., `string=?`, `char=?`).
- Use `eq?` or `eqv?` only if you are sure you know what you are doing.

Scheme: Strings & I/O
Console Output [1/2]

Print any value with `display`. String conversion is automatic. No trailing newline is printed. Print a newline with `newline`. Both of these return `void`, which does not print in the REPL.

```
> (display "Howdy thar!")
Howdy thar!
> (newline)

> (display #\A)
A
> (display '(42 #t (300)))
(42 #t (300))
> (display +)
#<procedure:+>
```

Scheme: Strings & I/O
Console Output [2/2]

To do multiple I/O calls in a single expression, use `begin`. This takes any number of arguments, evaluates them all, in order, and returns the value of the last one.

```
> (begin
    (display "dog")
    (display "food")
    (display "love")
  )
dogfoodlove
```

`begin` takes arbitrary expressions, not just those that do I/O.

Scheme: Strings & I/O
Console Interaction [1/5]

Read a line from the console with `read-line`. This takes no parameters. It returns the typed-in line with no trailing newline.

```
> (begin (display "Type something: ") (read-line))
Type something: Hello there!
Hello there!
```

Typed by user

How can we set a *local* variable to the return value of `read-line` in a procedure?

We use `let`.

```
(let  ; Locally bind vars to values in the expression
  (
    [VARIABLE1 VALUE1]
    …
    [VARIABLEn VALUEn]
  )
  ( EXPRESSION )
)
```

Scheme: Strings & I/O
Console Interaction [3/5]

TO DO

- Write a procedure to prompt for input, read a line, and then print it, with some explanation ("`Here is what you typed:`").

See `string.scm`.

Console Interaction [4/5]

We can repeat using a recursive procedure.

Alternatively, use `let` with a *name*.

```
(let NAME
   (
     [VARIABLE1  VALUE1]
     …
     [VARIABLEn  VALUEn]
   )
   ( EXPRESSION )
)
```

Within *EXPRESSION*, we can use *NAME* as a procedure taking *n* arguments. These become the values of *VARIABLE1* through *VARIABLEn* in that invocation of the procedure.

Scheme: Strings & I/O
Console Interaction [5/5]

TO DO

- Write a procedure that reads a series of numbers, until a blank line is entered, printing a running total after each. It should also respond in a reasonable way if the user types something other than a number.

See `string.scm`.