Module 3: New types of data

Readings: Sections 4 and 5 of HtDP.

A Scheme program applies functions to values to compute new values. These new values may in turn be supplied as arguments to other functions.

So far, we have seen values that are numbers, strings, and images.

In this module, we will see more different types of values.

In the next module, we will learn how to construct new types of values.

The data type boolean

A **Boolean value** is either true or false.

A **Boolean function** produces a Boolean value.

Scheme provides many built-in Boolean functions (for example, to do comparisons).

(= x y) is equivalent to determining whether "$x = y$" is true or false.

;; = : num num $\rightarrow$ boolean

Other types of comparisons

A **comparison** is a function that consumes two numbers and produces a Boolean value.

Other comparisons:

(< x y)

(> x y)

(<= x y)

(>= x y)

We can also compare strings using string=?, string<?, and so on.

Complex relationships

You may have learned in a math class how mathematical statements can be combined using the connectives AND, OR, NOT.

Scheme provides the corresponding and, or, not.

These are used to check complex relationships.

The statement "$3 \leq x < 7$" is the same as "$x \in [3, 7)$", and can be checked by evaluating

(and (<= 3 x) (< x 7)).

Some computational differences

The special forms and and or can each have two or more arguments.

The special form and has value true exactly when
all of its arguments have value true.

The special form or has value true exactly when
at least one of its arguments has value true.

The function not has value true exactly when
its one argument has value false.

The arguments of and and or are evaluated in order from left to right.

The evaluation stops as soon as the value can be determined.

Not all arguments may be evaluated.

```
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
(or (= x 0) (> (/ 100 x) 5))
```

Substitution rules for and

Here are the simplifications for application of and (slightly different from the textbook).

(and true exp . . .) yields (and exp . . .).

(and false exp . . .) yields false.

(and) yields true.

The last rule is needed when all arguments evaluate to true.

```
(define (good? str)
  (and (>= (string-length str) 3)
       (string=? "cat" (substring str 0 3))))
(good? "at")
⇒ (and (>= (string-length "at") 3)
       (string=? "cat" (substring "at" 0 3))))
⇒ (and (>= 2 3)
       (string=? "cat" (substring "at" 0 3)))
⇒ (and false (string=? "cat" (substring "at" 0 3)))
⇒ false
```

```
(define str "catch")
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and (>= (string-length "catch") 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and (>= 5 3)
     (string=? "cat" (substring str 0 3)))
⇒ (and true (string=? "cat" (substring str 0 3)))
⇒ (and (string=? "cat" (substring str 0 3)))
⇒ (and (string=? "cat" (substring "catch" 0 3)))
⇒ (and (string=? "cat" "cat")) ⇒ (and true) ⇒ (and) ⇒ true
```

Substitution rules for or

The rules for or are similar, but with the roles of true and false exchanged.

(or true exp . . .) yields true.

(or false exp . . .) yields (or exp . . .).

(or) yields false.

```
(define x 10)
(or (= x 0) (> (/ 100 x) 5))
⇒ (or (= 10 0) (> (/ 100 x) 5))
⇒ (or false (> (/ 100 x) 5))
⇒ (or (> (/ 100 x) 5))
⇒ (or (> (/ 100 10) 5))
⇒ (or (> 10 5))
⇒ (or true) ⇒ true
```

```
(define x 0)
(or (= x 0) (> (/ 100 x) 5))
⇒ (or (= 0 0) (> (/ 100 x) 5))
⇒ (or true (> (/ 100 x) 5)) ⇒ true
```

An example trace using and and or

(and (< 3 5) (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))

⇒ (and true (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))

⇒ (and (or (< 1 3) (= 1 3)) (or (> 1 5) (> 2 5)))

⇒ (and (or true (= 1 3)) (or (> 1 5) (> 2 5)))

⇒ (and true (or (> 1 5) (> 2 5)))

⇒ (and (or (> 1 5) (> 2 5)))

⇒ (and (or false (> 2 5)))

⇒ (and (or (> 2 5)))

⇒ (and (or false)) ⇒ (and (or))

⇒ (and false) ⇒ false

Predicates

A **predicate** is a Boolean function that determines if data is of a particular form.

Built-in predicates: e.g. even?, negative?, zero?, string?

User-defined (with full design recipe!):

```
(define (between? low high nbr)
  (and (< low nbr) (< nbr high)))

(define (can-drink? age)
  (>= age 19))
```

Conditional expressions

Sometimes expressions should take different values under different conditions.

- These use the special form cond.
- Each argument of cond is a question/answer pair.
- The question is a Boolean expression.
- The answer is a possible value of the conditional expression.

Example: taking the absolute value of x.

$$|x| = \begin{cases} -x & \text{when } x < 0 \\ x & \text{when } x \geq 0 \end{cases}$$

In Scheme, we can compute $|x|$ with the expression

```
(cond
  [(< x 0)  (- x)]
  [(>= x 0)    x])
```

- square brackets used by convention, for readability
- square brackets and parentheses are equivalent in the teaching languages (must be nested properly)
- abs is a built-in Scheme function

The general form of a conditional expression is

```
(cond
   [question1 answer1]
   [question2 answer2]
   ...
   [questionk answerk])
```

where questionk could be else.

The questions are evaluated in order; as soon as one evaluates to true, the corresponding answer is evaluated and becomes the value of the whole expression.

- The questions are evaluated in top-to-bottom order.
- As soon as one question is found that evaluates to true, no further questions are evaluated.
- Only one answer is ever evaluated, that is
  - the one associated with the first question that evaluates to true, or
  - the one associated with the else if that is present and reached.

Substitution rules for conds

There are three substitution rules: when the first expression is false, when it is true, and when it is else.

(cond [false . . .][exp1 exp2]. . .)

yields (cond [exp1 exp2]. . .).

(cond [true exp]. . .) yields exp.

(cond [else exp]) yields exp.

Example:

```
(define n 5)
(cond [(even? n) "even"] [(odd? n) "odd"])
⇒ (cond [(even? 5) "even"] [(odd? n) "odd"])
⇒ (cond [false "even"] [(odd? n) "odd"])
⇒ (cond [(odd? n) "odd"])
⇒ (cond [(odd? 5) "odd"])
⇒ (cond [true "odd"])
⇒ "odd"
```

```
(define (check-divide n)
  (cond [(zero? n) "Infinity"] [else (/ 1 n)]))
(check-divide 0)
⇒ (cond [(zero? 0) "Infinity"] [else (/ 1 0)])
⇒ (cond [true "Infinity"] [else (/ 1 0)])
⇒ "Infinity"
```

Nested conditionals

$$f(x) = \begin{cases} 0 & \text{when } x = 0 \\ \max\{\sin(x), 1/x\} & \text{when } x \neq 0 \end{cases}$$

We can use a conditional expression to compute the maximum:

```
(cond
  [(>= (/ 1 x)  (sin x)) (/ 1 x)]
  [(< (/ 1 x)  (sin x)) (sin x)])
```

```
(define (f x)
  (cond
    [(zero? x) 0]
    [else
     (cond
       [(>= (/ 1 x)  (sin x)) (/ 1 x)]
       [(< (/ 1 x)  (sin x)) (sin x)])
```

Flattening a nested conditional

```
(define (f x)
  (cond
    [(zero? x) 0]
    [(and (not (zero? x)) (>= (/ 1 x)  (sin x))) (/ 1 x)]
    [(and (not (zero? x)) (< (/ 1 x)  (sin x))) (sin x)]))
(define (f x)
  (cond
    [(zero? x) 0]
    [(>= (/ 1 x) (sin x)) (/ 1 x)]
    [else (sin x)]))
```

Conditional expressions can be used like any other expressions:

```
(define (my-fun x y)
  (+ (* x y)
     (cond
       [(even? x) x]
       [else y])))

(or (cond
      [(zero? x) (even? y)]
      [else (odd? x)])
    (integer? z))
```

Design recipe modifications

When we add to the language, we adjust the design recipe.

We add new steps and modify old steps.

If we don't mention a step, it is because it has not changed. It does not mean that it is no longer needed!

New step: data analysis

Modified steps: examples, tests

Design recipe modifications for conditional functions

Data analysis: figure out the different cases (possible outcomes).

Determine the subset of inputs that lead to each case.

Examples: should check interiors of intervals as well as boundary or borderline cases.

Definition:

Choose an ordering of the cases.

Determine questions to distinguish between cases.

Develop each question-answer pair one at a time.

Revisiting charges-for

Data analysis:

- outcomes: zero charge or charged per minute over free limit
- intervals: minutes below or above free limit
- question: compare minutes to free limit

Examples:

minutes below free limit

minutes above free limit

minutes equal to free limit

```
(define (charges-for minutes freelimit rate)
  (cond
    [(< minutes freelimit) 0]
    [(>= minutes freelimit) (* (- minutes freelimit) rate)]))

(define (cell-bill day eve)
 (+ (charges-for day day-free day-rate)
    (charges-for eve eve-free eve-rate)))
```

Bonus mark example

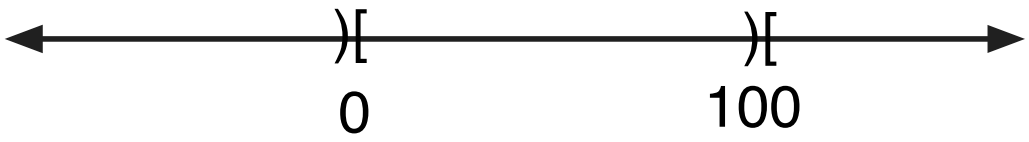
bonus determines bonus marks based on the difference between the sum of the first three assignments and the sum of the last three assignments.

bonus consumes the difference and produces the bonus:

- the difference for an increase of up to 100
- double the difference for an increase of 100 or more
- zero for no increase

For a difference diff, the cases are diff, (∗ 2 diff), and 0.

Here are in the intervals for each case:



Examples: 50, -50, 150, 0, 100, 125

```
(define (bonus diff)
  (cond
    [(and (<= 0 diff) (< diff 100)) diff]
    [(<= 100 diff) (* 2 diff)]
    [(< diff 0) 0]))
```

Nicer ordering (question reached if previous all false):

```
(define (bonus diff)
  (cond
    [(< diff 0) 0]
    [(< diff 100) diff]
    [else (* 2 diff)]))
```

Testing conditional expressions

Include at least one test for each answer, to exercise the code in each answer.

Ensure that each test is simple and direct, targeted to a particular purpose.

The purpose of each test should be clear.

Often it is appropriate to include tests at boundaries as well.

Unused code is highlighted in black.

Testing bonus marks

```
(define (bonus diff)
  (cond
    [(< diff 0) 0]
    [(< diff 100) diff]
    [else (* 2 diff)]))
;; Tests for bonus
(check-expect (bonus -50) 0)
(check-expect (bonus 50) 50)
(check-expect (bonus 150) 300)
```

Testing Boolean expressions

For and: one test case that produces true and one test case for each way of producing false.

For or: one test case that produces false and one test case for each way of producing true.

```
(and (>= (string-length str) 3)
     (string=? "cat" (substring str 0 3)))
```

"catch" (length at least three, starts with "cat"),
"at" (length less than three),
"dine" (length at least three, doesn't start with "cat").

Black-box tests and white-box tests

Black-box tests: based on different cases for what is consumed and produced, not on details of code. Good examples will suffice as black-box tests.

White-box tests: exercise all of the code, including

- at least each line of code (including thorough testing of boolean expressions), and
- each way that a question of a cond can be made true.

Use both: black-box before coding, white-box after coding.

Boolean tests

The textbook writes tests in this fashion:

```
(= (bonus -50) 0)
```

You can visually check tests by looking for trues.

These tests work outside of the teaching languages.

These will be useful in CS 116.

Substring checking

cat-start-or-end? asks if a string starts or ends with "cat".

Possible outcomes: true and false.

Inputs leading to true: starts with "cat", ends with "cat", starts and ends with "cat".

Questions to ask:

Does the string start with "cat"?

Does the string end with "cat"?

Developing predicates

```
(define (cat-start? str)
  (string=? "cat" (substring str 0 3)))

(define (cat-end? str)
  (string=? "cat" (substring str
    (- (string-length str) 3) (string-length str))))

(define (too-short? str)
  (> 3 (string-length str)))
```

Refined data analysis:

- Too short to contain "cat": produces false.
- Starts with "cat": produces true.
- Ends with "cat": produces true.
- Long enough but doesn't start or end with "cat": produces false.

Examples: "me", "caterpillar", "polecat", and "no cat here".

```
(define (cat-start-or-end? str)
  (cond
    [(too-short? str) false]
    [(cat-start? str) true]
    [(cat-end? str) true]
    [else false]))
;; Tests for cat-start-or-end?
(check-expect (cat-start-or-end? "me") false)
(check-expect (cat-start-or-end? "caterpillar") true)
(check-expect (cat-start-or-end? "polecat") true)
(check-expect (cat-start-or-end? "no cat here") false)
```

The data type symbol

Scheme allows one to define and use **symbols** with meaning to us (not to Scheme).

Symbols have a special marker to indicate that they are not function names or strings.

Syntax rule: a **symbol** starts with a single quote ’ followed by something obeying the rules for identifiers.

The symbol ’blue is a value just like 6, but it is more limited computationally.

Uses:

- A programmer can avoid using numbers to represent names of colours, or of planets, or of types of music.
- Symbols can be compared using the function symbol=?.

```
(define mysymbol 'blue)
(symbol=? mysymbol 'red) ⇒ false
```

Limitations:

No other comparisons or computations possible.

Symbols versus strings

Strings are **compound** data, as a string is a sequence of characters.

Symbols are **atomic** (indivisible), like numbers.

Comparing two symbols is more efficient than comparing two strings.

Symbols can't have spaces in them.

There are more built-in functions for strings.

When to use symbols vs. strings

Symbols:

- the number of labels needed (e.g. colours) is a small, fixed number, and
- the only computation needed is comparing labels for equality.

Strings:

- the set of values is more indeterminate, or
- more computation is needed (e.g. comparison in alphabetic order).

The data type character

Strings can be broken up into individual characters; later we will learn how to break strings apart and put them back together.

A character is an atomic data type.

Markers: #\space, #\1, #\a.

Built-in-functions: char<? to compare alphabetical order.

Built-in predicates: char-upper-case?.

The course Web page has more information.

A function that consumes a character might distinguish among upper-case letters, lower-case letters, numbers, white space, and punctuation marks.

```
(define (my-char-fun achar)
  (cond
    [(char-upper-case? achar) ...]
    [(char-lower-case? achar) ...]
    [(char-numeric? achar) ...]
    [(char-whitespace? achar) ...]
    [else ...]))
```

Mixed data

Sometimes a function will consume one of several types of data, or will produce one of several types of data.

```
;; check-divide: num → (union string num)
(define (check-divide n)
  (cond [(zero? n) "Infinity"] [else (/ 1 n)]))
```

Use the union notation for mixed data.

Generalized even

gen-even? consumes an integer, a symbol, or a string, producing true if the input is 'even, "even", or an even integer, and false otherwise.

What is the contract?

How many examples are needed to cover all cases?

A function that consumes mixed data contains a cond with one question for each type of data.

Scheme provides predicates to identify data types, such as number? and symbol?.

```
(define (gen-even? arg)
  (cond
    [(integer? arg) ...]
    [(symbol? arg) ...]
    [(string? arg) ...]))
```

Completing generalized even

```
(define (gen-even? info)
  (cond
    [(integer? info)
     (cond [(even? info) true] [else false])]
    [(symbol? info)
     (cond [(symbol=? info 'even) true] [else false])]
    [(string? info)
     (cond [(string=? info "even") true] [else false])]))
```

Simplifying generalized even

```
(define (gen-even? info)
  (cond
    [(integer? info) (even? info)]
    [(symbol? info) (symbol=? info 'even)]
    [(string? info) (string=? info "even")]))
```

General equality testing

There are equality predicates for each type: e.g. =, string=?, symbol=?, char=?.

The predicate equal? can be used to test the equality of two values which may or may not be of the same type.

equal? works for all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.

Generalized even using equal?

Using equal? we can eliminate checking if the input is a string or a symbol.

```
(define (gen-even? info)
  (cond
    [(integer? info) (even? info)]
    [(or (equal? info 'even) (equal? info "even")) true]
    [else false]))
```

Additions to syntax

Syntax rule: an **expression** can be:

- a *value*,
- a single *constant*,
- a *function application*,
- a *conditional expression*, or
- a Boolean expression.

Syntax rule: A **constant** is a number, a *symbol*, a *character*, a string, or a boolean value.

Design recipe for conditional functions

1. **Data analysis**. Figure out outcomes and inputs leading to each outcome.
2. **Contract** and **function header**.
3. **Purpose**.
4. **Examples**. Check interiors of intervals and boundaries.
5. **Body**. Order the cases. Determine questions. Develop each question-answer pair.
6. **Tests**. Include at least one test for each answer.
7. Run the program.

Goals of this module

You should be comfortable with these terms: Boolean value, Boolean function, comparison, predicate, compound, atomic.

You should be able to perform and combine comparisons to test complex conditions on numbers.

You should be able to trace programs using the substitution rules for and, or, and cond.

You should understand the syntax and use of a conditional expression.

You should use data analysis in the design recipe, and both black-box and white-box testing.

You should be able to write programs using symbols, characters, and mixed data, and know when to use symbols and when to use strings.

You should understand the (union . . .) notation and be able to use it in your own code.