How to Design Data (HtDD)

Help

Data definitions are a driving element in the design recipes.

A data definition establishes the represent/interpret relationship between information and data:

- Information in the program's domain is represented by data in the program.
- Data in the program can be interpreted as information in the program's domain.

A data definition must describe how to form (or make) data that satisfies the data definition and also how to tell whether a data value satisfies the data definition. It must also describe how to represent information in the program's domain as data and interpret a data value as information.

So, for example, one data definition might say that numbers are used to represent the `Speed` of a ball. Another data definition might say that numbers are used to represent the `Height` of an airplane. So given a number like 6, we need a data definition to tell us how to interpret it: is it a `Speed`, or a `Height` or something else entirely. Without a data definition, the 6 could mean anything.

> The first step of the recipe is to identify the inherent structure of the information.
>
> Once that is done, a data definition consists of four or five elements:
>
> 1. A possible **structure definition** (not until compound data)
> 2. A **type comment** that defines a new type name and describes how to form data of that type.
> 3. An **interpretation** that describes the correspondence between information and data.
> 4. One or more **examples** of the data.
> 5. A **template** for a 1 argument function operating on data of this type.
>
> In the first weeks of the course we also ask you to include a list of the **template rules** used to form the template.

What is the Inherent Structure of the Information?

One of the most important points in the course is that:

- the **structure of the information** in the program's domain determines the kind of data definition used,
- which in turn determines **the structure of the templates** and helps determine the function

examples (`check-expect`s),

- and therefore the **structure of much of the final program design**.

The remainder of this page lists in detail different kinds of data definition that are used to represent information with different structures. The page also shows in detail how to design a data definition of each kind. This summary table provides a quick reference to which kind of data definition to use for different information structures.

When the form of the information to be represented...	**Use a data definition of this kind**
is atomic	Simple Atomic Data
is numbers within a certain range	Interval
consists of a fixed number of distinct values	Enumeration
is comprised of 2 or more subclasses, at least one of which is not a distinct data item	Itemization
consists of two or more values that naturally belong together	Compound data
is naturally composed of different parts	References to other defined type
is of arbitrary (unknown) size	self-referential or mutually referential

Simple Atomic Data

Use simple atomic data **when the information to be represented is itself atomic in form**, such as the elapsed time since the start of the animation, the x coordinate of a car or the name of a cat.

```
;; Time is Natural
;; interp. number of clock ticks since start of game

(define START-TIME 0)
(define OLD-TIME 1000)

#;
(define (fn-for-time t)
  (... t))

;; Template rules used:
```

```
;;  - atomic non-distinct: Natural
```

Forming the Template

As noted below the template, it is formed according to the Data Driven Templates recipe using the right hand column of the atomic non-distinct rule.

Guidance on Data Examples and Function Example/Tests

One or two data examples are usually sufficient for simple atomic data.

When creating example/tests for a specific function operating on simple atomic data at least one test case will be required. Additional tests are required if there are multiple cases involved. If the function produces `Boolean` there needs to be at least a `true` and `false` test case. Also be on the lookout for cases where a number of some form is an interval in disguise, for example given a type comment like `Countdown is Natural`, in some functions `0` is likely to be a special case.

Intervals

Use an interval when the information to be represented is numbers within a certain range. `Integer[0, 10]` is all the integers from 0 to 10 inclusive; `Number[0, 10)` is all the numbers from 0 inclusive to 10 exclusive. (The notation is that [and] mean that the end of the interval includes the end point; (and) mean that the end of the interval does not include the end point.)

Intervals often appear in itemizations, but can also appear alone, as in:

```
;; Countdown is Integer[0, 10]
;; interp. the number of seconds remaining to liftoff
(define C1 10)  ; start
(define C2 5)   ; middle
(define C3 0)   ; end

#;
(define (fn-for-countdown cd)
  (... cd))

;; Template rules used:
;;  - atomic non-distinct: Integer[0, 10]
```

Forming the Template

As noted below the template, it is formed according to the Data Driven Templates recipe using the right hand column of the atomic non-distinct rule.

Guidance on Data Examples and Function Example/Tests

For data examples provide sufficient examples to illustrate how the type represents information. The three data examples above are probably more than is needed in that case.

When writing tests for functions operating on intervals be sure to test closed boundaries as well as midpoints. As always, be sure to include enough tests to check all other points of variance in behaviour across the interval.

Enumerations

Use an enumeration **when the information to be represented consists of a fixed number of distinct values**, such as colors, letter grades etc. In the case of enumerations it is sometimes redundant to provide an interpretation and nearly always redundant to provide examples. The example below includes the interpretation but not the examples.

```
;; LightState is one of:
;;  - "red"
;;  - "yellow"
;;  - "green"
;; interp. the color of a traffic light

;; <examples are redundant for enumerations>

#;
(define (fn-for-light-state ls)
  (cond [(string=? "red" ls) (...)]
        [(string=? "yellow" ls) (...)]
        [(string=? "green" ls) (...)]))
;; Template rules used:
;;  - one of: 3 cases
;;  - atomic distinct: "red"
;;  - atomic distinct: "yellow"
;;  - atomic distinct: "green"
```

Forming the Template

As noted below the template, it is formed according to the Data Driven Templates recipe as follows:

First, `LightState` is an enumeration with 3 cases, so the *one of rule* says to use a `cond` with 3 cases:

```
(define (fn-for-tlcolor ls)
  (cond [Q1 A1]
        [Q2 A2]
        [Q3 A3]))
```

In the first clause, `"red"` is a distinct atomic value, so the `cond` question column of the *atomic distinct rule* says Q1 should be `(string=? ls "red")`. The cond answer column says A1 should be `(...)`. So we have:

```
(define (fn-for-light-state ls)
  (cond [(string=? "red" ls) (...)]
        [Q2 A2]
        [Q3 A3]))
```

Then `"yellow"` and `"green"` are also distinct atomic values, so the final template is:

```
(define (fn-for-light-state ls)
  (cond [(string=? "red" ls) (...)]
        [(string=? "yellow" ls) (...)]
        [(string=? "green" ls) (...)]))
```

Guidance on Data Examples and Function Example/Tests

Data examples are redundant for enumerations.

Functions operating on enumerations should have (at least) as many tests as there are cases in the enumeration.

Large Enumerations

Some enumerations contain a large number of elements. A canonical example is `KeyEvent`, which is provided as part of big-bang. `KeyEvent` includes all the letters of the alphabet as well as other keys you can press on the keyboard. It is not necessary to write out all the cases for such a data definition. Instead write one or two, as well as a comment saying what the others are, where they are defined etc.

Defer writing templates for such large enumerations until a template is needed for a specific function. At that point include the specific cases that function cares about. Be sure to include an else clause in the template to handle the other cases. As an example, some functions operating on `KeyEvent` may only care about the space key and just ignore all other keys, the following would be an appropriate template for such functions.

```
#;
(define (fn-for-key-event kevt)
  (cond [(key=? " " kevt) (...)]
        [else
         (...)]))
;; Template formed using the large enumeration special case
```

The same is true of writing tests for functions operating on large enumerations. All the specially

handled cases must be tested, in addition one more test is required to check the else clause.

Itemizations

An itemization describes **data comprised of 2 or more subclasses, at least one of which is not a distinct data item**. (C.f. enumerations, where the subclasses are **all** distinct data items.) In an itemization the template is similar to that for enumerations: a cond with one clause per subclass. In cases where the subclass of data has its own data definition the answer part of the cond clause includes a call to a helper template, in other cases it just includes the parameter.

```
;; Bird is one of:
;;  - false
;;  - Number
;; interp. false means no bird, number is x position of bird

(define B1 false)
(define B2 3)

#;
(define (fn-for-bird b)
  (cond [(false? b) (...)]
        [(number? b) (... b)]))
;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: false
;;  - atomic non-distinct: Number
```

Forming the Template

As noted below the template, it is formed according to the Data Driven Templates recipe using the *one-of rule*, the *atomic distinct rule* and the *atomic non-distinct rule* in order.

Guidance on Data Examples and Function Example/Tests

As always, itemizations should have enough data examples to clearly illustrate how the type represents information.

Functions operating on itemizations should have at least as many tests as there are cases in the itemizations. If there are intervals in the itemization, then there should be tests at all points of variance in the interval. In the case of adjoining intervals it is critical to test the boundaries.

Itemization of Intervals

A common case is for the itemization to be comprised of 2 or more intervals. In this case functions operating on the data definition will usually need to be tested at all the boundaries of closed intervals and points between the boundaries.

```
;;; Reading is one of:
;;  - Number[> 30]
;;  - Number(5, 30]
;;  - Number[0, 5]
;; interp. distance in centimeters from bumper to obstacle
;;    Number[> 30]    is considered "safe"
;;    Number(5, 30]   is considered "warning"
;;    Number[0, 5]    is considered "dangerous"
(define R1 40)
(define R2 .9)

(define (fn-for-reading r)
  (cond [(< 30 r) (... r)]
        [(and (<  5 r) (<= r  30)) (... r)]
        [(<= 0 r 5) (... r)]))

;; Template rules used:
;;  one-of: 3 cases
;;  atomic non-distinct:  Number[>30]
;;  atomic non-distinct:  Number(5, 30]
;;  atomic non-distinct:  Number[0, 5]
```

As noted below the template, it is formed according to the Data Driven Templates recipe using the *one-of rule*, followed by 3 uses of the *atomic non-distinct rule*.

Compound data (structures)

Use structures when two or more values naturally belong together. The define-struct goes at the beginning of the data definition, before the types comment.

```
(define-struct ball (x y))
;; Ball is (make-ball Number Number)
;; interp. a ball at position x, y

(define BALL-1 (make-ball 6 10))

#;
(define (fn-for-ball b)
  (... (ball-x b)
       (ball-y b)))
;; Template rules used:
```

```
;;  - compound: 2 fields
;;  - atomic non-distinct: x field is Number
;;  - atomic non-distinct: y field is Number
```

The template above is formed according to the Data Driven Templates recipe using the compound rule. Then for each of the selectors, the result type of the selector (Number in the case of ball-x and ball-y) is used to decide whether the selector call itself should be wrapped in another expression. In this case, where the result types are primitive, no additional wrapping occurs. C.f. cases below when the reference rule applies.

Guidance on Data Examples and Function Example/Tests

For compound data definitions it is often useful to have numerous examples, for example to illustrate special cases. For a snake in a snake game you might have an example where the snake is very short, very long, hitting the edge of a box, touching food etc. These data examples can also be useful for writing function tests because they save space in each `check-expect`.

References to other data definitions

Some data definitions contain references to other data definitions you have defined (non-primitive data definitions). One common case is for a compound data definition to comprise other named data definitions. (Or, once lists are introduced, for a list to contain elements that are described by another data definition. In these cases the template of the first data definition should contain calls to the second data definition's template function wherever the second data appears. For example:

```
---assume Ball is as defined above---

(define-struct game (ball score))
;; Game is (make-game Ball Number)

;; interp. the current ball and score of the game

(define GAME-1 (make-game (make-ball 1 5 7 11) 2))

#;
(define (fn-for-game g)
  (... (fn-for-ball (game-ball g))
       (game-score g)))
;; Template rules used:
;;  - compound: 2 fields
;;  - reference: ball field is Ball
;;  - atomic non-distinct: score field is Number
```

In this case the template is formed according to the Data Driven Templates recipe by first using the *compound rule*. Then, since the result type of `(game-ball g)` is Ball, the *reference rule* is used to wrap the selector so that it becomes `(fn-for-ball (game-ball g))`. The call to `game-score` is not wrapped because it produces a primitive type.

Guidance on Data Examples and Function Example/Tests

For data definitions involving references to non-primitive types the data examples can sometimes become quite long. In these cases it can be helpful to define well-named constants for data examples for the referred to type and then use those constants in the referring from type. For example:

```
...in the data definition for Drop...
(define DTOP (make-drop 10 0))            ;top of screen
(define DMID (make-drop 20 (/ HEIGHT 2))) ;middle of screen
(define DBOT (make-drop 30 HEIGHT))       ;at bottom edge
(define DOUT (make-drop 40 (+ HEIGHT 1))) ;past bottom edge

...in the data definition for ListOfDrop...
(define LOD1 empty)
(define LOD-ALL-ON             (cons DTOP (cons DMID )))
(define LOD-ONE-ABOUT-TO-LEAVE (cons DTOP (cons DMID (cons DBOT empty))))
(define LOD-ONE-OUT-ALREADY    (cons DTOP (cons DMID (cons DBOT (cons DOUT empty)))))
```

In the case of references to non-primitive types the function operating on the referring type (i.e. `ListOfDrop`) will end up with a call to a helper that operates on the referred to type (i.e. `Drop`). Tests on the helper function should fully test that function, tests on the calling function may assume the helper function works properly.

Self-referential or mutually referential

When the **information in the program's domain is of arbitrary size**, a well-formed self-referential (or mutually referential) data definition is needed.

In order to be well-formed, a self-referential data definition must:

- (i) have at least one case without self reference (the base case(s))
- (ii) have at least one case with self reference

The template contains a base case corresponding to the non-self-referential clause(s) as well as one or more natural recursions corresponding to the self-referential clauses.

```
;; ListOfString is one of:
;;  - empty
;;  - (cons String ListOfString)
;; interp. a list of strings
```

```
(define LOS-1 empty)
(define LOS-2 (cons "a" empty))
(define LOS-3 (cons "b" (cons "c" empty)))

#;
(define (fn-for-los los)
  (cond [(empty? los) (...)]                    ;BASE CASE
        [else (... (first los)
                   (fn-for-los (rest los)))])) ;NATURAL RECURSION
;;             /
;;            /
;;       COMBINATION
;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons String ListOfString)
;;  - atomic non-distinct: (first los) is String
;;  - self-reference: (rest los) is ListOfString
```

In some cases a types comment can have both self-reference and reference to another type.

```
(define-struct dot (x y))
;; Dot is (make-dot Integer Integer)
;; interp. A dot on the screen, w/ x and y coordinates.
(define D1 (make-dot 10 30))
#;
(define (fn-for-dot d)
  (... (dot-x d) (dot-y d)))
;; Template rules used:
;;  - compound: 2 fields
;;  - atomic non-distinct: x field is Integer
;;  - atomic non-distinct: y field is type Integer


;; ListOfDot is one of:
;;  - empty
;;  - (cons Dot ListOfDot)
;; interp. a list of Dot
(define LOD1 empty)
(define LOD2 (cons (make-dot 10 20) (cons (make-dot 3 6) empty)))
#;
(define (fn-for-lod lod)
```

```
  (cond [(empty? lod) (...)]
        [else
         (... (fn-for-dot (first lod))
              (fn-for-lod (rest lod)))]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons Dot ListOfDot)
;;  - reference: (first lod) is Dot
;;  - self-reference: (rest lod) is ListOfDot
```

Guidance on Data Examples and Function Example/Tests

When writing data and function examples for self-referential data definitions always put the base case first. Its usually trivial for data examples, but many function tests don't work properly if the base case isn't working properly, so testing that first can help avoid being confused by a failure in a non base case test. Also be sure to have a test for a list (or other structure) that is at least 2 long.

Created Thu 21 Feb 2013 2:11 PM EST (UTC -0500)

Last Modified Thu 20 Jun 2013 12:38 PM EDT (UTC -0400)