Data Driven Templates

Help

Templates are the core structure that we know a function must have, independent of the details of its definition. In many cases the template for a function is determined by the type of data the function consumes. We refer to these as data driven templates. The recipe below can be used to produce a data driven template for any type comment.

For a given type `TypeName` the data driven template is:

```
(define (fn-for-type-name x)
  <body>)
```

Where `x` is an appropriately chosen parameter name (often the initials of the type name) and the body is determined according to the table below. To use the table, start with the type of the parameter, i.e. TypeName, and select the row of the table that matches that type. The first row matches only primitive types, the later rows match parts of type comments.

(Note that when designing functions that consume additional atomic parameters, the name of that parameter gets added after every `...` in the template. Templates for functions with additional complex paremeters are covered in Functions on 2 One-Of Data.)

Type of data	**`cond` question (if applicable)**	**Body or `cond` answer (if applicable)**
Atomic Non-Distinct • `Number` • `String` • `Boolean` • `Image` • interval like `Number[0, 10)` • etc.	Appropriate predicate • `(number? x)` • `(string? x)` • `(boolean? x)` • `(image? x)` • `(and (<= 0 x)` `(< x 10))` • etc.	Expression that operates on the parameter. `(... x)`
Atomic Distinct Value • `"red"` • `false` • `empty` • etc.	Appropriate predicate • `(string=? x "red")` • `(false? x)` • `(empty? x)` • etc.	Since value is distinct, parameter does not appear. `(...)`
One Of • enumerations		Cond with one clause per subclass of one of.

• enumerations • itemizations		`(cond [<question1> <answer1>]` `[<question2> <answer2>])` Where each question and answer expression is formed by following the rule in the question or answer column of this table for the corresponding case. A detailed derivation of a template for a one-of type appears below. It is permissible to use `else` for the last question for itemizations and large enumerations. Normal enumerations should not use else. Note that in a *mixed data itemization*, such as `;; Measurement is one of:` `;; - Number[-10, 0)` `;; - true` `;; - Number(0, 10]` the cond questions must be **guarded** with an appropriate type predicate. In particular, the first cond question for `Measurement` must be `(and (number? m)` `(<= -10 m)` `(< m 0))` where the call to `number?` guards the calls to <= and <. This will protect <= and < from ever receiving true as an argument.
Compound • `Position` • `Firework` • `Ball` • cons • etc.	Predicate from structure • `(posn? x)` • `(firework? x)` • `(ball? x)` • `(cons? x)` (often just else) • etc.	All selectors. • `(... (posn-x x) (posn-y x))` • `(... (firework-y x) (firework-color x))` • `(... (ball-x x) (ball-dx x))` • `(... (first x) (rest x))` • etc. Then consider the result type of each selector call and wrap the accessor expression appropriately using the table

		with that type. So for example, if after adding all the selectors you have: `(... (game-ball g) ;produces Ball` `(game-paddle g)) ;produces Paddle` Then, because both Ball and Paddle are non-primitive types (types that you yourself defined in a data definition) the reference rule (immediately below) says that you should add calls to those types' template functions as follows: `(... (fn-for-ball (game-ball g))` `(fn-for-paddle (game-paddle g)))`
Other Non-Primitive Type Reference	Predicate, usually from structure definition • `(firework? x)` • `(person? x)`	Call to other type's template function • `(fn-for-firework x)` • `(fn-for-person x)`
Self Reference		Form natural recursion with call to this type's template function: • `(fn-for-los (rest los))`

Mutual Reference
 	
Note: form and group all templates in mutual reference cycle together.
		Call to other type's template function:
		`(fn-for-lod (dir-subdirs d)`
		`(fn-for-dir (first lod))`

Producing the Template for an Example One Of Type

In many cases more than one of the above rules will apply to a single template. Consider this type comment:

```
;; Clock is one of:
;; - Natural
;; - false
```

and the step-by-step construction of the template for a function operating on `Clock`.

`(define (fn-for-clock c)`	`Clock` is a one of type with two subclasses (one of which is not

```
  (cond [Q A]
        [Q A]))

;; Template rules used:
;;  - one of: 2 cases
```
	distinct making it an itemization). The one of rule tells us to use a `cond`. The `cond` needs one clause for each subclass of the itemization.

```
(define (fn-for-clock c)
  (cond [(number? c) (...
 c)]
        [Q A]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic non-distinct
: Natural
```
	The `cond` questions need to identify each subclass of data. The `cond` answers need to follow templating rules for that subclasses data. In the first subclass, `Natural` is a non-distinct type; the *atomic non-distinct rule* tells us the question and answer as shown to the left.

```
(define (fn-for-clock c)
  (cond [(number? c) (...
 c)]
        [else
         (...)]))

;; Template rules used:
;;  - one of: 2 cases
;;  - atomic non-distinct
: Natural
;;  - atomic distinct: fa
lse
```
	In the second case `false` is an atomic distinct type, so the *atomic-distinct rule* gives us the question and answer. Since the second case is also the last case we can use `else` for the question.

Templates for Mutually Referential Types

The previous example doesn't cover the *mutual-reference rule* (), which says that in the case of mutually self-referential data definitions, when you template one function in the self-reference cycle you should **immediately template all the functions in the self-reference cycle**. So, for example, given:

```
(define-struct person (name subs))
;; Person is (make-person String ListOfPerson)
;; ListOfPerson is one of:
;;  - empty
;;  - (cons Person ListOfPerson)
```

Then if you need a template for a function operating on a `Person` (or a function operating on a `ListOfPerson`) you should immediately write a template for both functions, resulting in something like this:

```
#;
(define (fn-for-person p)
  (... (person-name p)
       (fn-for-lop (person-subs p))))  ;mutual recursion from mutual-reference

#;
(define (fn-for-lop lop)
  (cond [(empty? lop) ...]
        [else
         (... (fn-for-person (first lop))  ;mutual recursion from mutual-reference
              (fn-for-lop (rest lop)))]))  ;natural recursion from self-reference
```

(Note that producing that template will also involve using the atomic-distinct, atomic, one-of and compound rules.)

As with self-reference, its a good idea to draw a mutual-reference line on the data definition and ensure you have corresponding mutual recursion lines in your templates.

Testing

The principles above can also be used to understand how many tests a data definition implies. Simply put, the set of tests/examples should cover all cases, call all helper functions, involve all selectors, and avoid duplicated values.

Additional Design Rules for Helpers

During coding three additional guidelines suggest situations under which a helper function should be added:

1. Use a separate function for each difference between quantities in a problem.
2. If a subtask requires operating on arbitrary sized data a helper function must be used.
3. If a subtask involves special domain knowledge a helper function should be used.
4. In addition always keep the "one task per function" goal in mind. If part of a function you are designing seems to be a well-defined subtask put that into a helper function.

Created Thu 21 Feb 2013 2:08 PM EST (UTC -0500)

Last Modified Thu 12 Sep 2013 10:07 AM EDT (UTC -0400)