DESIGN RECIPES

In this course, we teach an approach to program design based on design recipes. Each recipe is applicable to certain problems, and systematizes the process of designing solutions to those problems.

There are three core recipes that are used most frequently. The templating recipes are used as part of the design of every data definition and function. Abstraction recipes are used to reduce redundancy in code.

You can also download a condensed version of some of the design recipes here for easy reference.

<table>
<tr><th rowspan="2">Core Recipes</th><th colspan="2">Templating</th><th rowspan="2">Abstraction</th></tr>
<tr><th>Data Driven</th><th>Control Driven</th></tr>
<tr><td>How to Design Functions (HtDF)
Design any function.</td><td>Data Driven Templates
Produce template for a data definition based on the form of the type comment.</td><td>Function Composition</td><td>From Examples
Produce an abstract function given two similar functions.</td></tr>
<tr><td>How to Design Data (HtDD)
Produce data definitions based on structure of the information to be represented.</td><td>2 One-of Data
Functions where 2 arguments have a one-of in their type comments.</td><td>Backtracking Search</td><td>From Type Comments
Produce a fold function given type comments.</td></tr>
<tr><td>How to Design Worlds (HtDW)
Produce interactive programs that use <code>big-bang</code>.</td><td rowspan="2"></td><td>Generative Recursion</td><td></td></tr>
<tr><td rowspan="2"></td><td>Accumulators</td><td>Using Abstract Functions</td></tr>
<tr><td colspan="2">Template Blending</td><td></td></tr>
</table>

Back to Design Recipes Table

How To Design Functions (HtDF)

The How to Design Functions (HtDF) recipe is a **design method** that enables systematic design of functions. We will use this recipe throughout the term, although we will enhance it as we go to solves more complex problems.

The HtDF recipe consists of the following steps:

1. Signature, purpose and stub.
2. Define examples, wrap each in `check-expect`.
3. Template and inventory.
4. Code the function body.
5. Test and debug until correct

NOTE:

- Each of these steps build on the ones that precede it. The signature helps write the purpose, the stub, and the check-expects; it also helps code the body. The purpose helps write the check-expects and code the body. The stub helps to write the check-expects. The check-expects help to code the body as well as to test the complete design.
- It is sometimes helpful to do the steps in a different order. Sometimes it is easier to write examples first, then do signature and purpose. Often at some point during the design you may discover an issue or boundary condition you did not anticipate, at that point go back and update the purpose and examples accordingly. But you should never write the function definition first and then go back and do the other recipe elements -- for some of you that will work for simple functions, but you will not be able to do that for the complex functions later in the course!
- Throughout the HtDF process be sure to "run early and run often". Run your program whenever it is well-formed. The more often you press run the sooner you can find mistakes. Finding mistakes one at a time is much easier than waiting until later when the mistakes can compound and be more confusing. Run, run, run!

Signature, purpose and stub.

Write the function signature, a one-line purpose statement and a function stub.

A signature has the type of each argument, separated by spaces, followed by `->`, followed by the type of result. So a function that consumes an image and produces a number would have the signature `Image -> Number`.

Note that the stub is a syntactically complete function definition that produces a value of the right type. If the type is Number it is common to use 0, if the type is String it is common to use "a" and so on. The value will not, in general, match the purpose statement. In the example below the stub produces 0, which is a Number, but only matches the purpose when double happens to be called with 0.

```
;; Number -> Number
;; produces n times 2

(define (double n)  0)  ; this is the stub
```

The purpose of the stub is to serve as a kind of scaffolding to make it possible to run the examples even before the function design is complete. With the stub in place check-expects that call the function can run. Most of them will fail of course, but the fact that they can run at all allows you to ensure that they are at least well-formed: parentheses are balanced, function calls have the proper number of arguments, function and constant names are correct and so on. This is very important, the sooner you find a mistake -- even a simple one -- the easier it is to fix.

Define examples, wrap each one in check-expect.

Write at least one example of a call to the function and the expected result the call should produce.

You will often need more examples, to help you better understand the function or to properly test the function. (If once your function works and you run the program some of the code is highlighted in black it means you definitely do not have enough examples.) If you are unsure how to start writing examples use the combination of the function signature and the data definition(s) to help you generate examples. Often the example data from the data definition is useful, but it does not necessarily cover all the important cases for a particular function.

The first role of an example is to help you understand what the function is supposed to do. If there are boundary conditions be sure to include an example of them. If there are different behaviours the function should have, include an example of each. Since they are examples first, you could write them in this form:

```
;; (double 0) should produce 0
;; (double 1) should produce 2
;; (double 2) should produce 4
```

When you write examples it is sometimes helpful to write not just the expected result, but also how it is computed. For example, you might write the following instead of the above:

```
;; (double 0) should produce (* 0 2)
;; (double 1) should produce (* 1 2)
;; (double 2) should produce (* 2 2)
```

While the above form satisfies our need for examples, DrRacket gives us a better way to write them, by enclosing them in check-expect. This will allow DrRacket to check them automatically when the function is complete. (In technical terms it will turn the examples into unit tests.)

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

(define (double n)  0)  ; this is the stub
```

The existence of the stub will allow you to run the tests. Most (or even all) of the tests will fail since the stub is returning the same value every time. But you will at least be able to check that the parentheses are balanced, that you have not misspelled function names etc.

Template and inventory

Before coding the function body it is helpful to have a clear sense of what the function has to work with -- what is the contents of your bag of parts for coding this function? The template provides this.

Once the How to Design Data Definitions (HtDD) recipe in introduced, templates are produced by following the rules on the Data Driven Templates web page. You should copy the template from the data definition to the function design, rename the template, and write a comment that says where the template was copied from. Note that the template is copied from the data definition for the consumed type, not the produced type.

For primitive data like numbers, strings and images the body of the template is simply `(... x)` where x is the name of the parameter to the function.

Once the template is done the stub should be commented out.

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

;(define (double n) 0) ; this is the stub

(define (double n)     ; this is the template
  (... n))
```

It is also often useful to add constant values which are extremely likely to be useful to the template body at this point. For example, the template for a function that renders the state of a world program might have an MTS constant added to its body. This causes the template to include an inventory of useful constants.

Code the function body

Now complete the function body by filling in the template.

Note that:

- the signature tells you the type of the parameter(s) and the type of the data the function body must produce
- the purpose describes what the function body must produce in English
- the examples provide several concrete examples of what the function body must produce
- the template tells you the raw material you have to work with

You should use all of the above to help you code the function body. In some cases further rewriting of examples might make it more clear how you computed certain values, and that may make it easier to code the function.

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

;(define (double n) 0) ; this is the stub

;(define (double n)    ; this is the template
;  (... n))

(define (double n)
  (* n 2))
```

Test and debug until correct

Run the program and make sure all the tests pass, if not debug until they do. Many of the problems you might have had will already have been fixed because of following the "run early, run often" approach. But if not, debug until everything works.

Back to Design Recipes Table

How To Design Data (HTDD)

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 items	Enumeration
is comprised of 2 or more subclasses, at least one of which is not a distinct item	Itemization

When the form of the information to be represented...	Use a data definition of this kind
consists of two or more items 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 items**, such as colors, letter grades etc. The data used for an enumeration could in principle be anything - strings, integers, images even. But we always use strings. 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 item**. (C.f. enumerations, where the subclasses are **all** distinct 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)     ;Number
       (ball-y b)))   ;Number
;; Template rules used:
;;  - compound: 2 fields
```

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) 2))

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

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)                  ;String
                   (fn-for-los (rest los)))]))  ;NATURAL RECURSION
;;             /
;;            /
;;       COMBINATION
;; Template rules used:
;;  - one of: 2 cases
;;  - atomic distinct: empty
;;  - compound: (cons String ListOfString)
;;  - 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)   ;Integer
       (dot-y d))) ;Integer
;; Template rules used:
;;  - compound: 2 fields

;; 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.

Back to Design Recipes Table

How To Design Worlds (HtDW)

The How to Design Worlds process provides guidance for designing interactive world programs using `big-bang`. While some elements of the process are tailored to `big-bang`, the process can also be adapted to the design of other interactive programs. The wish-list technique can be used in any multi-function program.

World program design is divided into two phases, each of which has sub-parts:

1. Domain analysis (use a piece of paper!)
    1. Sketch program scenarios
    2. Identify constant information
    3. Identify changing information
    4. Identify big-bang options
2. Build the actual program
    1. Constants (based on 1.2 above)
    2. Data definitions using HtDD (based on 1.3 above)
    3. Functions using HtDF
        1. main first (based on 1.3, 1.4 and 2.2 above)
        2. wish list entriesfor big-bang handlers
    4. Work through wish list until done

Phase 1: Domain Analysis

Do a domain analysis by hand-drawing three or more pictures of what the world program will look like at different stages when it is running.

Use this picture to identify constant information such as the height and width of screen, color of the background, the background image itself, the length of a firework's fuse, the image for a moving cat and so on.

Also identify changing information such as the position of a firework, the color of a light, the number in countdown etc.

Identify which `big-bang` options the program needs.

If your program needs to:	**Then it needs this option:**
change as time goes by (nearly all do)	`on-tick`
display something (nearly all do)	`to-draw`
change in response to key presses	`on-key`
change in response to mouse activity	`on-mouse`
stop automatically	`stop-when`

(There are several more options to `big-bang`. Look in the DrRacket help desk under `big-bang` for a complete list.)

Phase 2: Building the actual program

Structure the actual program in four parts:

1. Requires followed by one line summary of program's behavior
2. Constants
3. Data definitions
4. Functions

The program should begin with whatever require declarations are required. For a program using `big-bang` this is usually a require for `2htdp/universe` to get `big-bang` itself and a require for `2htdp/image` to get useful image primitives. This is followed by a short summary of the program's behavior (ideally 1 line).

The next section of the file should define constants. These will typically come directly from the domain analysis.

This is followed by data definitions. The data definitions describe how the world state - the changing information identified during the analysis - will be represented as data in the program. Simple world programs may have just a single data definition. More complex world programs have a number of data definitions.

The functions section should begin with the `main` function which uses `big-bang` with the appropriate options identified during the analysis. After that put the more important functions first followed by the less important helpers. Keep groups of closely related functions together.

Template for a World Program

A useful template for a world program, including a template for the main function and wish list entries for tick-handler and to-draw handler is as follows. To use this template replace WS with the appropriate type for your changing world state. You may want to give the handler functions more descriptive names and you should definitely give them all a more descriptive purpose.

```
(require 2htdp/image)
(require 2htdp/universe)

;; My world program  (make this more specific)

;; =================
;; Constants:


;; =================
;; Data definitions:

;; WS is ... (give WS a better name)



;; =================
;; Functions:

;; WS -> WS
;; start the world with ...
;;
(define (main ws)
  (big-bang ws                    ; WS
            (on-tick   tock)      ; WS -> WS
            (to-draw   render)    ; WS -> Image
            (stop-when ...)       ; WS -> Boolean
            (on-mouse  ...)       ; WS Integer Integer MouseEvent -> WS
            (on-key    ...)))     ; WS KeyEvent -> WS

;; WS -> WS
;; produce the next ...
;; !!!
(define (tock ws) ...)


;; WS -> Image
;; render ...
;; !!!
(define (render ws) ...)
```

Depending on which other big-bang options you are using you would also end up with wish list entries for those handlers. So, at an early stage a world program might look like this:

```
(require 2htdp/universe)
(require 2htdp/image)

;; A cat that walks across the screen.

;; Constants:

(define WIDTH  200)
(define HEIGHT 200)

(define CAT-IMG (circle 10 "solid" "red")) ; a not very attractive cat

(define MTS (empty-scene WIDTH HEIGHT))


;; Data definitions:

;; Cat is Number
;; interp. x coordinate of cat (in screen coordinates)
(define C1 1)
(define C2 30)

#;
(define (fn-for-cat c)
  (... c))


;; Functions:

;; Cat -> Cat
;; start the world with initial state c, for example: (main 0)
(define (main c)
  (big-bang c                          ; Cat
            (on-tick   tock)           ; Cat -> Cat
            (to-draw   render)))       ; Cat -> Image

;; Cat -> Cat
;; Produce cat at next position
;!!!
(define (tock c) 1)  ;stub

;; Cat -> Image
;; produce image with CAT-IMG placed on MTS at proper x, y position
; !!!
(define (render c) MTS)
```

Note that we are maintaining a wish listof functions that need to be designed. The way to maintain the wish list is to just write a signature, purpose and stub for each wished-for function, also label the wish list entrywith `!!!` or some other marker that is easy to search for. That will help you find your unfilled wishes later.

Forming wish list entries this way is enough for `main` (or other functions that call a wished for function) to be defined without error. But of course `main` (and other such functions) will not run properly until the wished for functions are actually completely designed.

As you design the program remember to run early and run often. The sooner you can run the program after writing anything the sooner you can find any small mistakes that might be in it. Fixing the small mistakes earlier makes it easier to find any harder mistakes later.

Key and Mouse Handlers

The `on-key` and `on-mouse` handler function templates are handled specially. The `on-key` function is templated according to its second argument, a `KeyEvent`, using the large enumeration rule. The `on-mouse` function is templated according to its `MouseEvent` argument, also using the large enumeration rule. So, for example, for a key handler function that has a special behaviour when the space key is pressed but does nothing for any other key event the following would be the template:

```
(define (handle-key ws ke)
  (cond [(key=? ke " ") (... ws)]
        [else
         (... ws)]))
```

Similarly the template for a mouse handler function that has special behavior for mouse clicks but ignores all other mouse events would be:

```
(define (handle-mouse ws x y me)
  (cond [(mouse=? me "button-down") (... ws x y)]
        [else
         (... ws x y)]))
```

For more information on the `KeyEvent` and `MouseEvent` large enumerations see the DrRacket help desk.

Back to Design Recipes Table

Data Driven Templates

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. `(...)`

Type of data	cond question (if applicable)	Body or cond answer (if applicable)

One Of
• enumerations
• itemizations
		Cond with one clause per subclass of one of.
		
		`(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.

Type of data	**cond question (if applicable)**	**Body or cond answer (if applicable)**

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))`

Type of data	cond question (if applicable)	Body or cond answer (if applicable)

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)`
`  (cond [Q A]`
`  [Q A]))`
 	
`;; Template rules used:`
`;;  - one of: 2 cases`
	`Clock` is a one of type with two subclasses (one of which is not 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:
false
```

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.

Back to Design Recipes Table

Functions On 2 One-Of Data

This page outlines the variations in the normal HtDF recipe when designing a function that consumes 2 data that have a one-of in their type comments. Examples of functions for which this applies include functions with the following signatures:

```
;; ListOfString ListOfString -> Boolean
;; ListOfString ListOfString -> ListOfString
;; ListOfString BinaryTree -> Boolean
;; ListOfNatural FamilyTree -> ListOfString
```

For the purpose of this explanation, assume that the goal is to design a function that consumes two `ListOfString` and produces true if the strings in the first list are equal to the corresponding strings in the second list. If that is true and the second list is longer than the first the function should produce true; if the second list is shorter it always produces false.

The first three steps of the recipe - signature, purpose and stub - are unchanged.

```
;; ListOfString ListOfString -> Boolean
;; produce true if lsta is a prefix of lstb

(define (prefix=? lsta lstb) false)
```

But at this point the next step is to form a **cross-product of type comments** table as follows. The row labels of the table are the cases of the one-of type comment for one argument (perhaps

the first), and the column labels are the cases of the one-of type comment for the other argument.

```
;; CROSS PRODUCT OF TYPE COMMENTS TABLE
;;
;;                                    lstb
;;                            empty           (cons String LOS)
;;                                            |
;; l   empty                                  |
;; s                          ---------------------------------
;; t   (cons String LOS)                      |
;; a                                          |
```

In this case, where both arguments have 2 cases in their one-of type comments, the cross-product formed has 4 cells (2 * 2 = 4). The next step of the process is to use the cross product table to help form at least as many tests as there are cells. The upper left cell describes a scenario where both `lsta` and `lstb` are empty. The lower left cell is where `lsta` is non-empty but `lstb` is empty and so on. The lower right cell is where both lists are non-empty, and this case requires more than one test. So we end up with:

```
(check-expect (prefix=? empty empty) true)
(check-expect (prefix=? empty (list "a" "b")) true)
(check-expect (prefix=? (list "a") empty) false)

(check-expect (prefix=? (list "a") (list "b")) false)
(check-expect (prefix=? (list "a") (list "a")) true)
(check-expect (prefix=? (list "a" "b") (list "a" "x")) false)
(check-expect (prefix=? (list "a" "b") (list "a" "b")) true)
```

We can now use the tests to help fill in the contents of the table cells, indicating what the function should do in each case. Any cell requiring a more complex code answer, such as the lower right one in this case, need not be coded perfectly with correct syntax, but should give a good idea of what the code must do.

```
;; CROSS PRODUCT OF TYPE COMMENTS TABLE
;;
;;                                    lstb
;;                            empty           (cons String LOS)
;;                                            |
;; l   empty                    true           |  true
;; s                          ---------------------------------
;; t   (cons String LOS)        false         |  (and <firsts are string=?>
;; a                                          |       <rests are prefix=?>)
```

Now comes the most fun step. We look for a way to simplify the table by identifying cells that have the same answer. In this case the entire first row produces `true`, so we can simplify the table by combining the two cells in the first row into a single cell:

```
;; CROSS PRODUCT OF TYPE COMMENTS TABLE
;;
;;                                      lstb
;;                              empty           (cons String LOS)
;;                                        |
;; l   empty                                 true
;; s                              ---------------------------------
;; t   (cons String LOS)            false      |  (and <firsts are string=?>
;; a                                           |       <rests are prefix=?>)
```

Now we are almost done. The next step is to code the function body directly from the table. This in effect intertwines templating with coding of details. Because the simplified table has only three cells, we know that the body of the function will be a three case `cond`. For the first question we always pick the largest cell, in this case the top row. The question needs to be true of the entire combined cell, so in this case the question is `(empty? lsta)`. The answer in this case is just `true`.

For the next case of the `cond` we pick the lower left cell. At this point, we know that the top row has been handled, so we only need a question that distinguishes the remaining cells apart. In this case `(empty? lstb)` distinguishes the lower left from the lower right cell. The answer in this case is `false`.

In larger tables this process continues until you get to the last cell, at which point the question can be `else`.

In cells that involve natural recursion, the natural recursion can be formed by applying the normal rules for handling self-reference. In this case the template for the `cond` answer in the third `cond` case is:

```
              (... (first lsta)
                   (first lstb)
                   (prefix=? (rest lsta) ...)
                   (prefix=? ... (rest lstb)))
```

After filling that in we end up with:

```
(define (prefix=? lsta lstb)
  (cond [(empty? lsta) true]
        [(empty? lstb) false]
        [else
         (and (string=? (first lsta) (first lstb))
              (prefix=? (rest lsta)
                        (rest lstb)))]))
```

In general, when designing a function on 2 one of data it is a good idea to include the cross-product table in the design.

Back to Design Recipes Table

Function Composition

Use function composition when a function must perform two or more distinct and complete operations on the consumed data. For example:

- A function that must sort and layout a list of images. First it must sort the complete list and then lay it out. It cannot sort and layout each image one at a time.
- A function that must advance a list of raindrops and then remove the ones that have left the screen. First it must advance all the drops and then remove the ones that have advanced too far. (With difficulty this could be done in a single pass through the list of drops, but it is much more cumbersome to do that way.)

When using function composition the normal template for the function is discarded, and the body of the function has two or more function compositions. So in the case of `arrange-images` the function design would look like this:

```
;; ListOfImage -> Image
;; arrange images left to right in increasing order of size
(check-expect (arrange-images (list I1 I3 I2))
              (beside I1 I2 I3 BLANK))

(define (arrange-images loi)
  (layout-images (sort-images loi)))
```

Which we read as saying "first sort the images, and then layout the sorted list". At the point this is written wish list entries would be created for `layout-images` and `sort-images` unless those functions already existed.

Tests for a function that uses function composition should be selected to ensure that the function is calling all the appropriate functions and composing them properly. For example, assuming that images `I1`, `I2` and `I3` are in increasing order of size, then this test alone would not be adequate:

```
(check-expect (arrange-images (list I1 I2 I3))
              (beside I1 I2 I3 BLANK))
```

Because a faulty implementation of `arrange-images` that just calls `layout-images` would pass. Instead a test like the original one above is needed, to ensure that both `sort-images` and `layout-images` are called. But note that the tests for arrange-images do not themselves need to fully test both composed functions. They only need to test the composition. That is why `arrange-images` does not absolutely have to have a base case test. (Although it wouldn't hurt it to have one.)

Back to Design Recipes Table

Backtracking Search

The template for backtracking is:

```
(define (fn-for-x x)
  (... (fn-for-lox (x-subs x))))

(define (fn-for-lox lox)
  (cond [(empty? lox) false]
        [else
          (if (not (false? (fn-for-x (first lox))))  ;is first child
successful?
              (fn-for-x (first lox))                 ;if so produce that
              (fn-for-lox (rest lox)))]))            ;or try rest of
children
```

Note that this template incorporates the template for an n-ary tee, where the tree nodes have type x and x-subs produces the children of a tree node given the node. The backtracking works as commented above. Once we have local expressions we tend to write the backtracking search template as follows:

```
(define (backtracking-fn x)
  (local [(define (fn-for-x x)
            (... (fn-for-lox (x-subs x))))

          (define (fn-for-lox lox)
            (cond [(empty? lox) false]
                  [else
                   (local [(define try (fn-for-x (first lox)))] ;try first
child
                     (if (not (false? try))
;successful?
                         try                                    ;if so
produce that
                         (fn-for-lox (rest lox))))]))]          ;or try
rest of children

    (fn-for-x x)))
```

Back to Design Recipes Table

Generative Recursion

The template for generative recursion is:

```
(define (genrec-fn d)
  (cond [(trivial? d) (trivial-answer d)]
        [else
         (... d
              (genrec-fn (next-problem d)))]))
```

Back to Design Recipes Table

Accumulators

There are three general ways to use an accumulator in a recursive function (or set of mutually recursive functions):

1. To preserve context otherwise lost in structural recursion.
2. To make a function tail recursive by preserving a representation of the work done so far (aka a result-so-far accumulator).
3. To make a function tail recursive by preserving a representation of the work remaining to do (aka a worklist accumulator).

The same basic recipe covers all three forms of accumulators. The main example shown here is a context preserving accumulator.

The basic recipe and context preserving accumulators.

Signature, purpose, stub and examples

Design of the function begins normally, with signature, purpose, stub and examples.

```
;; (listof X) -> (listof X)
;; produce list formed by keeping the 1st, 3rd, 5th and so on elements of
lox
(check-expect (skip1 (list "a" "b" "c" "d")) (list "a" "c"))
(check-expect (skip1 (list 0 1 2 3 4)) (list 0 2 4))

(define (skip1 lox) empty) ;stub
```

Templating

The template step is a 3 part process. The first step is to template normally according to the rules for structural recursion, i.e. template according to the (listof X) parameter.

```
(define (skip1 lox)
  (cond [(empty? lox) (...)]
        [else
         (... (first lox)
              (skip1 (rest lox)))]))
```

The next step is to encapsulate that function in an outer function and local. As part of this step give the outer function parameter a different name than the inner function parameter. Note that if you are working with multiple mutually recursive functions they call get wrapped in a single outer function.

```
(define (skip1 lox0)
  (local [(define (skip1 lox)
            (cond [(empty? lox) (...)]
                  [else
                   (... (first lox)
                        (skip1 (rest lox)))]))]

    (skip1 lox0)))
```

Now add the accumulator parameter to the inner function. In addition, add ... or more substantial template expressions in each place that calls the inner function. During this step treat the accumulator parameter as atomic.

```
(define (skip1 lox0)
  (local [(define (skip1 lox acc)
            (cond [(empty? lox) (... acc)]
                  [else
                   (... acc
                        (first lox)
                        (skip1 (rest lox)
                               (... acc (first lox))))]))]

    (skip1 lox0 ...)))
```

Accumulator type, invariant and examples

The next step is to work out what information the accumulator will represent and how it will do that. Will the accumulator serve to represent some context that would otherwise be lost to structural recursion? Or, to support tail recursion will it represent some form of result so far? Or will it represent a work list of some sort. In many cases this is clear before reaching this stage of the design process. In other cases examples can be used to work this out. But in all cases examples are useful to work out exactly how the accumulator will represent the information.

In this case we need to know, in each recursive call to the inner function, whether the current first item in the list should be kept or skipped. There are many ways to represent this, but one simple way is to use a natural that represents how far into the original list (lox0) we have traveled. If we assume the call to the top-level definition of skip1 is (skip1 (list 0 1 2 3 4)), then the progression of calls to the internal skip1 would be as follows:

```
(skip1 (list 0 1 2 3 4) 1)       the 0 is the 1st element of lox0 (using 1
based indexing)
(skip1 (list   1 2 3 4) 2)       the 1 is the 2nd element
(skip1 (list     2 3 4) 3)       the 2 is the 3rd
(skip1 (list       3 4) 4)       the 3 is the 4th
(skip1 (list         4) 5)       the 4 is the 5th
(skip1 (list          ) 6)
```

Note that the accumulator value is not constant, but it always represents the position of the current (first lox) in the original list lox0. (You can see here why we renamed the parameter to the outer function, it makes it easier to describe the relation between the original value lox0 and

the value in each recursive call lox.)

These examples of the progression of calls to the internal recursive function(s) allow us to work out clearly the accumulator type, as well as its *invariant*, which describes what is constant about the accumulator as it changes in other words, what property it always represents. In this case the type is Natural, and the invariant is the 1 based index of (first lox) in lox0. So when the accumulator is an odd number we will keep (first lox) in the result, when it is even we will skip (first lox).

```
(define (skip2 lox0)
  ;; acc is Natural; how many elements of lox to keep before next skip
  ;; (skip2 (list 0 1 2 3 4) 1)
  ;; (skip2 (list   1 2 3 4) 0)
  ;; (skip2 (list     2 3 4) 1)
  ;; (skip2 (list       3 4) 0)
  ;; (skip2 (list         4) 1)
  ;; (skip2 (list          )
  (local [(define (skip2 lox acc)
            (cond [(empty? lox) (... acc)]
                  [else
                   (... acc
                        (first lox)
                        (skip2 (rest lox)
                               (... acc (first lox))))]))]

    (skip2 lox0 ...)))
```

Complete the code

At this point the function definition can be completed by using the signature, purpose, examples, accumulator type, invariant and examples to fill in the details. When doing so, note three distinct aspects of coding with the accumulator invariant. *Initializing* the accumulator happens in the trampoline, and involves providing an initial value of the accumulator that satisfies the invariant. *Exploiting* the invariant involves counting on the accumulator to always represent what the accumulator describes. *Preserving* the invariant happens in recursive calls to the function where a (possibly) new value is provided for the accumulator argument. Preserving the invariant means making sure that the value provided in the recursive call satisfies the invariant.

```
(define (skip1 lox0)
  ;; acc is Natural; 1 based index of (first lox) in lox0
  ;; (skip1 (list 0 1 2 3 4) 1)    ;0 should be kept
  ;; (skip1 (list   1 2 3 4) 2)    ;1 should be skipped
  ;; (skip1 (list     2 3 4) 3)    ;0 should be kept
  ;; (skip1 (list       3 4) 4)    ;1 should be skipped
  ;; (skip1 (list         4) 5)    ;0 should be kept
  ;; (skip1 (list          ) 6)
  (local [(define (skip1 lox acc)
            (cond [(empty? lox) empty]
                  [else
                   (if (even? acc)
                       (skip1 (rest lox) (add1 acc))
                       (cons (first lox)
                             (skip1 (rest lox) (add1 acc))))]))]

    (skip1 lox0 1)))
```

Tail Recursion

To make the function tail recursive, all recursive calls must be in tail position. For functions that operate on flat structures (data that has only one reference cycle), this can be accomplished by using an accumulator to represent build up information about the final result through the series of recursive calls. We often name these accumulators rsf but it is worth noting that some functions require more than one result so far accumulator (see average).

Making functions that operate on data with more than one cycle in the graph (such as arbitrary-arity trees) usually requires the use of an accumulator to build up the data that still remains to be operated on. This is a worklist accumulator, often called todo.

Back to Design Recipes Table

Template Blending

To understand template blending its important to understand that templates are what we know about the core of a function (or set of functions) before we get to the details. Data driven (or structural recursion) templates are that, backtracking templates are that, and generative recursion templates are that.

In some cases we know more than one thing about the core structure of a function (or set of functions). In the sudoku-solver for example three different templates apply to the solve functions.

- **arbitrary-arity tree** - we consider each board to have a set of next boards formed by filling the first empty cell with the numbers from 1 - 9. So this forms an arbitrary-arity tree (the arity is actually [0, 9]).
- **generative recursion** - while each board has a set of next boards that set is included in the representation of the board. Instead those next boards have to be generated. So we have a

generated arbitrary-arity tree.

- **backtracking search** - in addition we need to do a backtracking search over the arbitrary-arity tree.

In template blending we take multiple templates that contribute to the structure of a function (or functions) and combine them together.

We have the following for the `solve` function:

```
;; Board -> Board or false

;; produce a solution for bd; or false if bd is unsolvable

;; Assume: bd is valid

(check-expect (solve BD4) BD4s)

(check-expect (solve BD5) BD5s)

(check-expect (solve BD7) false)



(define (solve bd) false) ;stub
```

Now let's start with the template for arbitrary-arity tree. Remember that the template for an arbitrary-arity tree involves a mutual recursion. In this case, we would have a function that consumes Board (ie. `solve--bd`) and calls another function (ie. `solve--lobd`) that does something to the (listof Board) that is supposed to come with the Board (a Board doesn't actually come with a (listof Board) but we will deal with it using generative recursion later). In order to complete the mutual recursion, we need to call `solve--bd` inside the function `solve--lobd`. So now we have the following template:

```
(define (solve bd)
  (local [(define (solve--bd bd)
            (... (solve--lobd (bd-subs bd))))


          (define (solve--lobd lobd)
            (cond [(empty? lobd) (...)]
                  [else
                   (... (solve--bd (first lobd))
                        (solve--lobd (rest lobd)))]))]

    (solve--bd bd)))
```

Keep in mind that the `bd-subs` selector doesn't exist because Board does not keep a (listof Board). We need to deal with it using generative recursion.

Let's blend this template with the generative recursion template. The generative recursion template looks like this:

```
(define (genrec-fn d)
  (if (trivial? d)
      (trivial-answer d)
      (... d
           (genrec-fn (next-problem d)))))
```

Since we know that `solve--bd` must generate a (listof Board) to pass to `solve--lobd`, we must blend the template into the `solve--bd` function. Then we would have (the red part was changed):

```
(define (solve bd)

  (local [(define (solve--bd bd)

            (if (solved? bd)

                bd

                (solve--lobd (next-boards bd))))



          (define (solve--lobd lobd)

            (cond [(empty? lobd) (...)]

                  [else

                   (... (solve--bd (first lobd))

                        (solve--lobd (rest lobd)))]))]



    (solve--bd bd)))
```

Notice the following:

- `bd-subs` is changed to `next-boards` because it suggests that we are generating new boards.
- `trivial?` from the generative recursion template is changed to `solved?` because the trivial case (ie. the case where the recursion should stop at) is when the board is solved.
- `trivial-answer` from the generative recursion template is omitted because in this case, the trivial answer is the board that is solved, which is represented by `bd`.

Lastly, we need to blend in the template for backtracking search. The template is:

```
(define (backtracking-fn x)

  (local [(define (fn-for-x x)

            (... (fn-for-lox (x-subs x))))



          (define (fn-for-lox lox)

            (cond [(empty? lox) false]

                  [else

                   (local [(define try (fn-for-x (first lox)))] ;try first
child

                     (if (not (false? try))
;successful?

                         try                                    ;if so
produce that

                         (fn-for-lox (rest lox))))]))]          ;or try
rest of children



    (fn-for-x x)))
```

We need to blend the template for `fn-for-lox` from above into the function `solve-lobd`. The changes are in red:

```
(define (solve bd)
  (local [(define (solve--bd bd)
            (if (solved? bd)
                bd
                (solve--lobd (next-boards bd))))


          (define (solve--lobd lobd)
            (cond [(empty? lobd) false]
                  [else
                   (local [(define try (solve--bd (first lobd)))]
                        (if (not (false? try))
                            try
                            (solve--lobd (rest lobd))))]))]


    (solve--bd bd)))
```

The template allows the `solve--lobd` function to do the following:

- If we have reached the end of the (listof Board) (ie. if `lobd` is empty - the base case), then produce false, meaning that the (listof Board) has no solution.
- If the (listof Board) is not empty, then try to solve the first board in the list (ie. `(solve--bd (first lobd))`).
- If the outcome of the try is not false (ie. `(not (false? try))`), then produce that outcome because it means the board has been solved.
- If the outcome is false, then recurse and try the rest of the list (ie. `(solve--lobd (rest lobd))`).

Now we have the complete template below. The arbitrary-arity tree template is underlined in blue. The generative recursion template is in red. And the backtracking search template is in green.

```
(define (solve bd)

  (local [(define (solve--bd bd)

            (if (solved? bd)

                bd

                (solve--lobd (next-boards bd)))


          (define (solve--lobd lobd)

            (cond [(empty? lobd) false]

                  [else

                   (local [(define try (solve--bd (first lobd)))]

                        (if (not (false? try))

                            try

                            (solve--lobd (rest lobd))))]))]


    (solve--bd bd)))
```

The next step to completing the function is to implement the two helper functions - `solved?` and `next-boards`.

Back to Design Recipes Table

Abstraction From Examples

We design abstract functions in the opposite order of the normal HtDF recipe. We always want to do the easiest thing first, and with the abstract function design processes getting the working function definition is the easiest thing to do. In fact going through the recipe in the opposite order exactly goes from easiest to hardest.

1. Identify two or more fragments of highly repetitive code. In general these can be expressions that appear within functions, or they can be entire functions. The rest of this recipe is tailored to the case where entire functions have been chosen.
2. Arrange the two functions so that it is easy to see them at the same time.
3. Identify one or more points where the functions differ (points of variance). Do not count differences in function names or parameter names as points of variance.

4. Copy one function definition to make new one
   - give the new function a more general name
   - add a new parameter for each point of variance
   - update any recursive calls to use new name and add new parameters in recursive calls
   - use the appropriate new parameter at each point of variability
   - rename other parameters to be more abstract (lon to lox for example)
5. Adapt tests from original functions to new abstract function
   - be sure to test variability
   - attempt to test behavior of the abstract function beyond that exercised by the two examples
6. Develop an appropriately abstract purpose based on the examples.
7. Develop an appropriate signature for the abstract function; in many cases the signature will include type parameters.
8. Rewrite the body of the two original functions to call the abstract function.

Back to Design Recipes Table

Abstraction From Type Comments

We design abstract functions in the opposite order of the normal HtDF recipe. We always want to do the easiest thing first, and with the abstract function design processes getting the working function definition is the easiest thing to do. In fact going through the recipe in the opposite order exactly goes from easiest to hardest.

1. if there are templates for mutually recursive functions first encapsulate them in a single template with local
2. replace each ... in the templates with a new parameter; for (...) remove the parens
3. develop examples (check-expects)
4. develop abstract purpose from examples
5. develop abstract signature from concrete examples

Let's now go through generating a fold function for `(listof X)`. Here is the type comment for `(listof X)`:

```
;; ListOfX is one of:

;; - empty

;; - (cons X ListOfX)
```

We can generate the following template based on this type comment:

```
(define (fn-for-lox lox)

  (cond [(empty? lox) (...)]

        [else

         (... (first lox)

              (fn-for-lox (rest lox)))]))
```

Since there is no mutual recursion in the template, we can skip step 1. According to step 2, we need to replace each `...` with a new parameter. We also need to rename the function to `fold`. Here is what we would have after step 2:

```
(define (fold fn b lox)

  (cond [(empty? lox) b]

        [else

         (fn (first lox)

             (fold fn b (rest lox)))]))
```

Note that we called the `...` in place of the base case is named to `b`, and the `...` in place of the function is named to `fn`.

For step 3, we should write some examples to test for the function and also find out what we can do with the fold function:

```
; sum of the numbers in the list

(check-expect (fold + 0 (list 1 2 3)) 6)



; product of the numbers in the list

(check-expect (fold * 1 (list 1 2 3)) 6)



; append the strings in the list

(check-expect (fold string-append "" (list "a" "bc" "def")) "abcdef")



; sum of the areas of the images in the list

(check-expect (local [(define (total-area i a)

                        (+ (* (image-width i)

                              (image-height i))

                           a))]

                (fold total-area 0 (list (rectangle 20 40 "solid" "red")

                                         (right-triangle 10 20 "solid"
"red")

                                         (circle 20 "solid" "red"))))

              (+ (* 20 40) (* 10 20) (* 40 40)))
```

Step 4 requires us to develop an abstract purpose from the examples. In this example, we can just write:

```
;; the abstract fold function for (listof X)
```

In the last step, we need to determine the abstract signature from the examples that we have written. The more diverse the examples that we have, the easier it is to come up with the correct signature.

From the function defintion for `fold`, we can first reason the following for the signature of `fold`:

- the parameter `lox` is of the type (listof X)
- the parameter `b` is of the same type as what the `fold` function produces because it is the base case
- the predicate function `fn` takes in two arguments
- the predicate function `fn` produces the same type as what the `fold` function produces
- the first parameter for the predicate function `fn` is of the type X
- the second parameter for the predicate function `fn` is of the same type as what the `fold` function produces

From the above reasonings, we can narrow the signature down to:

```
(X ??? -> ???) ??? (listof X) -> ???
```

Note that all of the `???` are of the same type according to the above reasoning. Now the question is whether `???` has to be the same type as X or can be something else (ie. Y).
From the fourth example in step 3, we can see that the signature for the predicate function `total-area` is:

```
Image Number -> Number
```

In this case, the X is Image and the `???` is Number. This is an example of `???` being something different than X. Therefore, we can conclude that the abstract signature for `fold` is:

```
(X Y -> Y) Y (listof X) -> Y
```

Back to Design Recipes Table

Using Abstract Functions

The template for using a built-in abstract function like filter is:

```
;; (listof Number) -> (listof Number)

;; produce only positive? elements of lon

;;
```

(define (only-positive lon) (filter ... lon))

Now we note that the type of lon is (listof Number); and the signature of filter is `(X -> Boolean) (listof X) -> (listof X)`. This means that that signature of the function passed to filter is `(Number -> Boolean)` so we can further decorate the template as follows:

```
(define (only-positive lon)

          ;(Number -> Boolean)

  (filter ...                   lon))
```