How to Design Programs

How to Design Programs (HDP) is a CS1 curriculum with a data-driven approach to program design. HDP teaches students to develop schemas (code templates) for functions from the structure of the input data. These templates traverse the data, leaving holes to fill to compute particular functions.

In the following example, the code in blue is the template for processing a list; the code in red is specific to the function.

```
(define (adding-machine L)
  (cond [(empty? L) 0]
        [(equal? "F" (first L)) (+ 1 (count-matches (rest L)))]
        [else (count-matches (rest L) ...)]))
```

Example: (count-matches (list "P" "F" "P" "F" "F" "J")) should yield 3

Benefit of HDP: traversals and data variants handled “automatically” in template, helps student focus on computation specific to the problem

How does this fit with existing models of how students develop programs?

Rist’s Focal Expansion Model

Rist’s focal expansion model highlights two states novices enter when they encounter a programming problem:

- Plan Retrieval
  - Novice retrieves the known solution and implements top-down, with limited adaption to the new problem.

- Plan Creation
  - Novice creates a new plan starting from a code fragment for a sub-computation (focus/focal computation). Novice expands code around the focus bottom-up to implement a solution.

Hypothesis on how HDP fits:
1. In retrieval mode, similar to Rist’s model
2. In creation mode, students default to the HDP-prescribed templates

Observation from pilot study: HDP can hinder successful plan creation when problems require decomposition beyond the template.

Investigating Novice Programmers’ Plan Composition Strategies
Francisco Enrique Vicente G. Castro, fgcastro@wpi.edu
Advisor: Kathi Fisler
Department of Computer Science, WPI

How to Design Programs (HDP) is a CS1 curriculum with a data-driven approach to program design. HDP teaches students to develop schemas (code templates) for functions from the structure of the input data. These templates traverse the data, leaving holes to fill to compute particular functions.

In the following example, the code in blue is the template for processing a list; the code in red is specific to the function.

```scheme
(define (average L)
  (/ (sum L) (length L)))
```

Example:

```
; Return the count of strings in a list that match the string, "F"
(define (count-matches L)
  (cond [(empty? L) 0]
       [(equal? "F" (first L)) (+ 1 (count-matches (rest L)))]
       [else (count-matches (rest L) ...)]))
```

Example: (count-matches (list "P" "F" "P" "F" "F" "J")) should yield 3

The basic HDP template doesn’t yield a natural decomposition of Rainfall. We don’t want to lose its data-driven focus when appropriate, however.

**Question:** Can we extend HDP’s data-driven approach to help novices decompose problems?

Adding Machine

Design a program called `adding-machine` that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

**Sample input data:**
```scheme
(list 3 7 0 5 6 0 0 9) should yield (list 10 5 6)
```

**Data principles to notice:**
1. Irrelevant data at end and (2) structured data has been flattened out
2. Clean and structured data:
   ```scheme
   (list (list 3 7) (list 5) (list 6))
   ```

**Remaining work on clean and structured data:**
produce list of sums of sublists

**Must write:**
- Function(s) to transform input data into clean and structured data: straight-up HDP function(s)
- Function(s) to produce list of sums of sublists: straight-up HDP function(s)

```scheme
(define (adding-machine L)
  (sum-lists (parse (truncator L))))
```

**Our pilot study recorded HDP students as they worked on this problem. None discovered the idea of data transformation even though it makes the core function straightforward. Data from another study shows that this problem is also difficult for students trained in traditional, imperative programming.**

Sample Problems

**Rainfall**

Design a program called `rainfall` that consumes a list of numbers representing daily rainfall amounts. Rainfall may contain the number -999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first -999 (if it shows up). There may be negative numbers other than -999 in the list.

**Sample input data:**
```scheme
(list -2 -5 -4 -999 8) should yield 3
```

**Data principles to notice:**
1. Data has “noise” and (2) irrelevant data at end
2. Cleaned-out data:
   ```scheme
   (list 2 4)
   ```

**Remaining work on clean data:**
compute its average

**Must write:**
- Function(s) for data-cleaning: straight-up HDP function(s) (whether done individually or simultaneously)
- Function(s) to compute the average (not straight-up HDP, see next section)

**Note:** If the same data principles come up often, students should begin to retrieve the code that performs them (see work by Soloway, Rist)

Leveraging Examples for Decomposition

**What if there is no cleaning or data restructuring to do?**
Leverage examples to help students discover (need for) decomposition.

**1. Example:**
   ```scheme
   (average (list 1 2 3 4 5)) should yield: 3
   ```

**2. Example:**
   ```scheme
   (average (list 2 3 4 5)) should yield: 3.5
   ```

**3. Question:** Can we easily combine 1 and 3.5 to get 3?

**4. Test case:**
   ```scheme
   (average (list 1 2 3 4 5)) should produce: (/ (+ (list 1 2 3 4 5)) 5)
   ```

**Straight-up HDP function**

**Length operator**

**Overall Goal:** Help students learn reliable techniques for developing correct programs.

**This work:** Extending HDP to teach problem decomposition. A data-driven approach might avoid decomposing on-the-fly while coding. Problem-level decomposition seems preferable.

**Main Research Questions:**
1. How do we help novices recognize when a problem needs decomposition beyond a basic data traversal?
2. How do we help them identify suitable decompositions?

**Current Research Focus:**
1. Can students learn to recognize and use data transformation patterns?
   a. Can students figure out what computation remains after data transformation?
2. Do enhanced examples help novices identify suitable decompositions?