Module 8: Compound data: structures

Readings: Sections 6 and 7 of HtDP.

- Sections 6.2, 6.6, 6.7, 7.4 are optional readings; they use the obsolete `draw.ss` teachpack.
- The teachpacks `image.ss` and `world.ss` are more useful.
- Note that none of these particular teachpacks will be used on assignments or exams.

New types defined using fixed-length lists

Recall that we can use data definitions to define new types, grouping related pieces of information together into a fixed length list. For example,

```
;; A Card is a (list Nat (anyof 'hearts 'diamonds 'spades 'clubs))
;; requires
;; * the first value (rank) is between 1 and 13, inclusive
```

We can use first and second to retrieve the rank and suit of a Card value, respectively.

Using fixed length lists and data definitions to define new types is very helpful. It allows us to use built-in list operations to process Card values.

A disadvantage is that code is not very readable unless we define new helper functions to retrieve individual values, such as

```
;; rank: Card → Nat
(define (rank c) (first c))
```

In this module, we will introduce a different way to define new types in Racket which is generally more readable than lists: structures.

Compound data

A **structure** is a way of "bundling" several pieces of data together to form a single "package".

We can

- create functions that consume and/or produce structures, and
- define our own structures, automatically getting ("for free") functions that create structures and functions that extract data from structures.

A new type

Suppose we want to design a program for a card game such as poker or cribbage. Before writing any functions, we have to decide on how to represent data.

For each card, we have a suit (one of hearts, diamonds, spades, and clubs) and a rank (for simplicity, we will consider ranks as integers between 1 and 13). We can create a new structure with two fields using the following **structure definition**.

```
(define-struct card (rank suit))
```

Using the Card type

Once we have defined our new type, we can:

- Create new values using the **constructor** function make-card

```
(define h5 (make-card 5 'hearts))
```

- Retrieve values of the individual fields using the **selector** functions card-rank and card-suit

(card-rank h5) $\Rightarrow$ 5

(card-suit h5) $\Rightarrow$ 'hearts

We can also

- Check if a value is of type Card using the **type predicate** function card?

(card? h5) $\Rightarrow$ true

(card? "5 of hearts") $\Rightarrow$ false

Once the new structure card has been defined, the functions make-card, card-rank, card-suit, card? are created by Racket. We do not have to write them ourselves.

We have grouped all the data for a single card into one value, and we can still retrieve the individual pieces of information.

More information about Card

The structure definition of Card does not provide all the information we need to use the new type properly. We will still need a **data definition** to provide additional information about the types of the different field values.

```
(define-struct card (rank suit))
;; A Card is a
;;      (make-card Nat (anyof 'hearts 'diamonds 'spades 'clubs))
;; requires
;;        rank is between 1 and 13, inclusive
```

Functions using Card values

```
;; (pair? c1 c2) produces true if c1 and c2 have the same rank,
;;    and false otherwise
;; pair?: Card Card → Bool
(define (pair? c1 c2) (= (card-rank c1) (card-rank c2)))

;; (one-better c) produces a Card, with the same suit as c, but
;;    whose rank is one more than c (to a maximum of 13)
;; one-better: Card → Card
(define (one-better c)
    (make-card (min 13 (+ 1 (card-rank c))) (card-suit c)))
```

Posn structures

A Posn (short for Position) is a built-in structure that has two **fields** containing numbers intended to represent x and y coordinates. We might want to use a Posn to represent coordinates of a point on a 2-D plane, positions on a screen, or a geographical position.

This structure definition is built-in. We'll use the following data definition.

```
;; A Posn is a (make-posn Num Num)
```

Built-in functions for Posn

;; make-posn: Num Num → Posn

;; posn-x: Posn → Num

;; posn-y: Posn → Num

;; posn?: Any → Bool

Examples of use

```
(define myposn (make-posn 8 1))
```

(posn-x myposn) ⇒ 8

(posn-y myposn) ⇒ 1

(posn? myposn) ⇒ true

Substitution rules and simplified values

For any values a and b

(posn-x (make-posn a b)) $\Rightarrow$ a

(posn-y (make-posn a b)) $\Rightarrow$ b

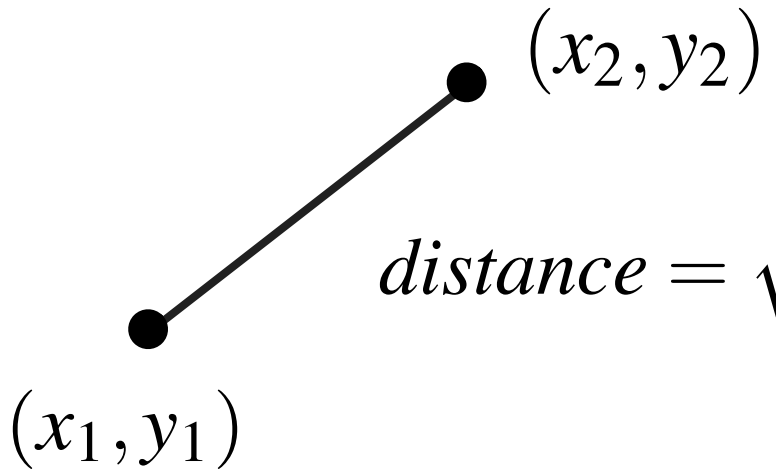
The make-posn you type is a function application.

The make-posn DrRacket displays indicates that the value is of type posn.

(make-posn (+ 4 4) (− 2 1)) yields (make-posn 8 1), which cannot be further simplified.

Similar rules apply to our newly defined card structure as well.

Example: point-to-point distance



$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The function distance

```
;; (distance posn1 posn2) produces the Euclidean distance
;;    between posn1 and posn2.
;; distance: Posn Posn → Num
;; example:
(check-expect (distance (make-posn 1 1) (make-posn 4 5)) 5)

(define (distance posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn1) (posn-x posn2)))
           (sqr (- (posn-y posn1) (posn-y posn2))))))
```

Function that produces a Posn

```
;; (scale point factor) produces the Posn that results when the fields
;;    of point are multiplied by factor
;; scale: Posn Num → Posn
;; Examples:
(check-expect (scale (make-posn 3 4) 0.5) (make-posn 1.5 2))
(check-expect (scale (make-posn 1 2) 1) (make-posn 1 2))

(define (scale point factor)
  (make-posn (* factor (posn-x point))
             (* factor (posn-y point))))
```

When we have a function that consumes a number and produces a number, we do not change the number we consume.

Instead, we make a new number.

The function scale consumes a Posn and produces a new Posn.

It doesn't change the old one.

Instead, it uses make-posn to make a new Posn.

Structure definitions for new structures

Structure definition: code defining a structure, and resulting in constructor, selector, and type predicate functions.

```
(define-struct sname (field1 field2 field3))
```

Writing this once creates functions that can be used many times:

- **Constructor**: make-sname
- **Selectors**: sname-field1, sname-field2, sname-field3
- **Predicate**: sname?

Design recipe modifications

Data analysis and design: design a data representation that is appropriate for the information handled in our function.

Determine which structures are needed and define them. Include

- a structure definition (code) and
- a data definition (comment).

Place the structure and data definitions immediately after the file header, before your constants and functions.

Structure for Movie information

Suppose we want to represent information associated with movies, that is:

- the name of the director
- the title of the movie
- the duration of the movie
- the genre of the movie (sci-fi, drama, comedy, etc.)

```
(define-struct movieinfo (director title duration genre))
;; A MovieInfo is a (make-movieinfo Str Str Nat Str)
;; where:
;;      director is the name of director of the movie,
;;      title is the name of movie,
;;      duration is the length of the movie in minutes,
;;      genre is the genre (type or category) of the movie.
```

Note: If all the field names for a new structure are self-explanatory, we will usually omit the field-by-field descriptions.

The structure definition gives us:

- Constructor make-movieinfo
- Selectors movieinfo-director, movieinfo-title, movieinfo-duration, and movieinfo-genre
- Predicate movieinfo?

```
(define et-movie
   (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi"))
```

(movieinfo-duration et-movie) $\Rightarrow$ 115

(movieinfo? 6) $\Rightarrow$ false

Templates and data-directed design

As we've seen with lists, one of the main ideas of the HtDP text is that the form of a program often mirrors the form of the data.

We will continue to derive templates from the data definition for each new structure type we define, and then will apply it many times in writing functions that consume that type of data.

A template for MovieInfo

The template for a function that consumes a structure selects every field in the structure, though a specific function may not use all the selectors.

```
;; movieinfo-template: MovieInfo → Any
(define (movieinfo-template info)
    (...(movieinfo-director info)...
        (movieinfo-title info)...
        (movieinfo-duration info)...
        (movieinfo-genre info)...))
```

An example

```
;; (correct-genre oldinfo newg) produces a new MovieInfo
;;    formed from oldinfo, correcting genre to newg.
;; correct-genre: MovieInfo Str → MovieInfo
;; example:
(check-expect
  (correct-genre
      (make-movieinfo "Spielberg" "ET" 115 "Comedy")
      "Sci-Fi")
  (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi"))
```

The function correct-genre

We use the parts of the template that we need, and add a new parameter.

```
(define (correct-genre oldinfo newg)
  (make-movieinfo (movieinfo-director oldinfo)
                  (movieinfo-title oldinfo)
                  (movieinfo-duration oldinfo)
                  newg))
```

We could have done this without a template, but the use of a template pays off when designing more complicated functions.

Additions to syntax for structures

The special form (define-struct sname (field1 . . . fieldn)) defines the structure type sname and automatically defines a constructor function, selector functions for each field, and a type predicate function.

A **value** is a number, a string, a boolean, a list, or is of the form (make-sname v1 . . . vn) for values v1 through vn.

Additions to semantics for structures

The substitution rule for the ith selector is:

(sname-fieldi (make-sname v1 ... vi ... vn)) $\Rightarrow$ vi

The substitution rules for the type predicate are:

(sname? (make-sname v1 ... vn)) $\Rightarrow$ true

(sname? V) $\Rightarrow$ false for a value V of any other type.

An example using posns

Recall the definition of the function scale :

```
(define (scale point factor)
  (make-posn (* factor (posn-x point))
             (* factor (posn-y point))))
```

Then we can make the following substitutions:

```
(define myposn (make-posn 4 2))
(scale myposn 0.5)
⇒ (scale (make-posn 4 2) 0.5)
⇒ (make-posn
   (* 0.5 (posn-x (make-posn 4 2)))
   (* 0.5 (posn-y (make-posn 4 2))))
⇒ (make-posn
   (* 0.5 4)
   (* 0.5 (posn-y (make-posn 4 2))))
```

$\Rightarrow$ (make-posn 2 (* 0.5 (posn-y (make-posn 4 2))))

$\Rightarrow$ (make-posn 2 (* 0.5 2))

$\Rightarrow$ (make-posn 2 1)

Since (make-posn 2 1) is a value, no further substitutions are needed.

Another example

```
(define mymovie (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
(correct-genre mymovie "Fantasy")
⇒ (correct-genre
(make-movieinfo "Reiner" "Princess Bride" 98 "War") "Fantasy")
⇒ (make-movieinfo
(movieinfo-director (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
"Fantasy")
```

```
⇒ (make-movieinfo
"Reiner"
(movieinfo-title (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
"Fantasy")
⇒ (make-movieinfo
"Reiner"
"Princess Bride"
(movieinfo-duration (make-movieinfo "Reiner" "Princess Bride" 98 "War"))
"Fantasy")
⇒ (make-movieinfo "Reiner" "Princess Bride" 98 "Fantasy")
```

Design recipe for compound data

Do this *once per new structure type:*

Data analysis and design: Define any new structures needed for the problem. Write structure and data definitions for each new type (include right after the file header).

Template: Create one template for each new type defined, and use for each function that consumes that type. Use a generic name for the template function and include a generic contract.

Do the usual design recipe steps for *every function:*

Purpose: Same as before.

Contract and requirements: Can use both built-in data types and defined structure names.

Examples: Same as before.

Function Definition: To write the body, expand the template based on the examples.

Tests: Same as before. Be sure to capture all cases.

Design recipe example

Suppose we wish to create a function total-length that consumes information about a TV series, and produces the total length (in minutes) of all episodes of the series.

Data analysis and design.

```
(define-struct tvseries (title eps len-per))
;; A TVSeries is a (make-tvseries Str Nat Nat)
;; where
;;        title is the name of the series
;;        eps is the total number of episodes
;;        len-per is the average length (in minutes) for one episode
```

The structure definition gives us:

- Constructor make-tvseries
- Selectors tvseries-title, tvseries-eps, and tvseries-len-per
- Predicate tvseries?

The data definition tells us:

- types required by make-tvseries
- types produced by tvseries-title, tvseries-eps, and tvseries-len-per

Templates for TVSeries

We can form a template for use in any function that consumes a single TVSeries:

```
;; tvseries-template: TVSeries → Any
(define (tvseries-template show)
  (... (tvseries-title show) ...
       (tvseries-eps show) ...
       (tvseries-len-per show) ...))
```

You might find it convenient to use constant definitions to create some data for use in examples and tests.

```
(define murdoch (make-tvseries "Murdoch Mysteries" 168 42))
(define friends (make-tvseries "Friends" 236 22))
(define fawlty (make-tvseries "Fawlty Towers" 12 30))
```

Mixed data and structures

Consider writing functions that use a streaming video file (movie or tv series), which does not require any new structure definitions.

```
(define-struct movieinfo (director title duration genre))
;; A MovieInfo is a (make-movieinfo Str Nat Str)
(define-struct tvseries (title eps len-per))
;; A TVSeries is a (make-tvseries Str Nat Nat)

;; A StreamingVideo is one of:
;; * a MovieInfo or
;; * a TVSeries.
```

The template for StreamingVideo

The template for mixed data is a cond with one question for each type of data.

```
;; streamingvideo-template: StreamingVideo → Any
(define (streamingvideo-template info)
  (cond [(movieinfo? info) ...]
        [else ...]))
```

We use type predicates in our questions.

Next, expand the template to include more information about the structures.

```
;; StreamingVideo-template: StreamingVideo → Any
(define (streamingvideo-template info)
  (cond [(movieinfo? info)
         (... (movieinfo-director info) ...
              (movieinfo-title info) ...
              (movieinfo-duration info) ...
              (movieinfo-genre info) ...) ]
        [else
         (... (tvseries-title info) ...
              (tvseries-eps info) ...
              (tvseries-len-per info) ...) ]))
```

An example: StreamingVideo

```
;; (streamingvideo-title info) produces title of info
;; streamingvideo-title: StreamingVideo → Str
;; Examples:
(check-expect (streamingvideo-title
    (make-movieinfo "Spielberg" "ET" 115 "Sci-Fi")) "ET")
(check-expect (streamingvideo-title
    (make-tvseries "Friends"  236 22)) "Friends")
(define (streamingvideo-title info) ...)
```

The definition of streamingvideo-title

```
(define (streamingvideo-title info)
  (cond
    [(movieinfo? info) (movieinfo-title info)]
    [else (tvseries-title info)]))
```

Reminder: anyof types

If a consumed or produced value for a function can be one of a restricted set of types or values, we will use the notation

(anyof type1 type2 ... typeK v1 ... vT)

For example, if we hadn't defined StreamingVideo as a new type, we could have written the contract for streamingvideo-title as

;; streamingvideo-title: (anyof MovieInfo TVSeries) $\rightarrow$ Str

A nested structure

```
(define-struct doublefeature (first second start-hour))
;; A DoubleFeature is a
;; (make-doublefeature MovieInfo MovieInfo Nat),
;; requires:
;;       start-hour is between 0 and 23, inclusive
```

An example of a DoubleFeature is

```
(define classic-movies
  (make-doublefeature
    (make-movieinfo "Welles" "Citizen Kane" 119 "Drama")
    (make-movieinfo "Kurosawa" "Rashomon" 88 "Mystery")
    20))
```

- Develop the function template.
- What is the title of the first movie?
- Do the two movies have the same genre?
- What is the total duration for both movies?

Structures containing lists

Suppose we store the name of a server along with a list of tips collected.

How might we store the information?

```
(define-struct server (name tips))
;; A Server is a (make-server Str (listof Num))
;; requires:
;;       numbers in tips are non-negative
```

We form templates for a server and for a list of numbers.

```
(define (server-template s)
  (... (server-name s) ...
       (lon-template (server-tips s)) ...))

(define (lon-template alon)
  (cond
    [(empty? alon) ...]
    [else (... (first alon) ... (lon-template (rest alon)) ...)]))
```

Note: We may choose to use abstract list functions instead of using lon-template when writing the function.

The function big-tips consumes a server s and a number smallest and produces the server formed from s by removing tips smaller than smallest.

```
(define (big-tip-list alon smallest)
  (filter (lambda (tip) (<= smallest tip)) alon))

(define (big-tips s smallest)
  (make-server (server-name s) (big-tip-list (server-tips s) smallest)))
```

Lists of structures

Suppose we wish to store marks for all the students in a class.

How do we store the information for a single student?

```
(define-struct student (name assts mid final))
;; A Student is a (make-student Str Num Num Num)
;; requires:
;;      assts, mid, final are between 0 and 100
```

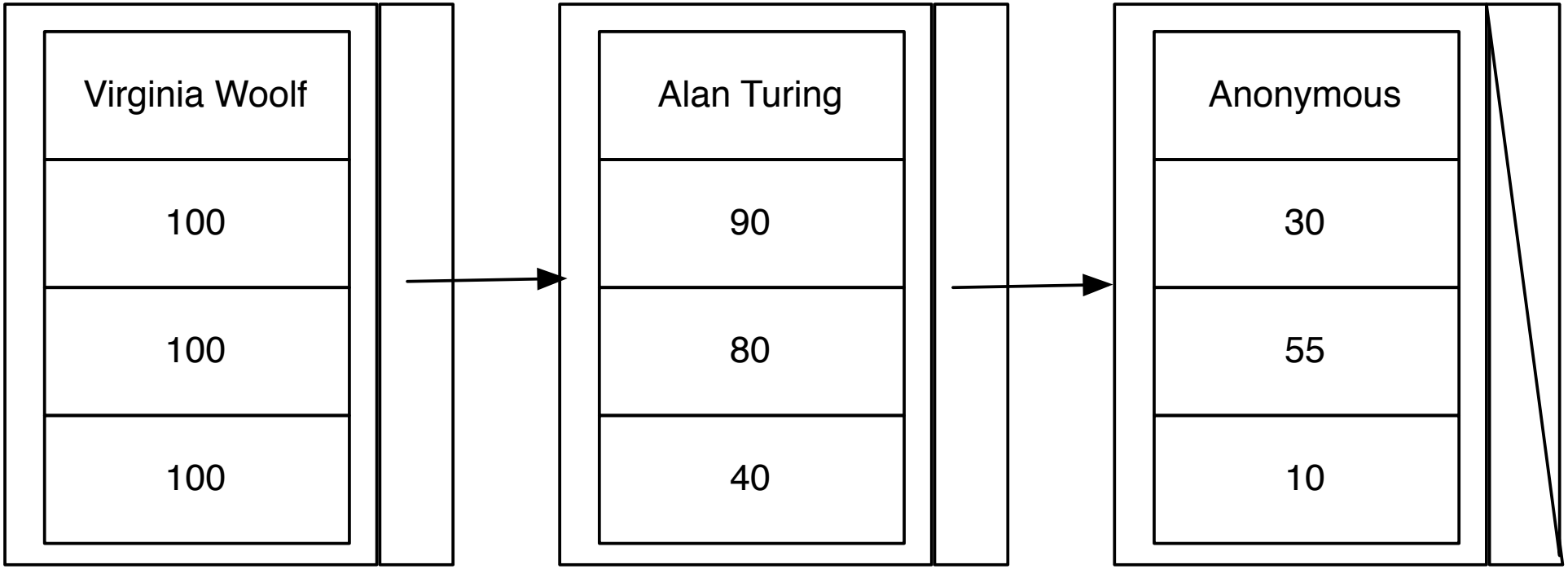
How do we store information for all the students?

```
;; A (listof Student) is one of:
;; * empty
;; * (cons Student (listof Student))
```

Example:

```
(cons (make-student "Virginia Woolf" 100 100 100)
   (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty)))
```

Visualization



A list of students

```
(define myslist
  (cons (make-student "Virginia Woolf" 100 100 100)
    (cons (make-student "Alan Turing" 90 80 40)
      (cons (make-student "Anonymous" 30 55 10) empty))))
```

What are the values of these expressions?

- `(first myslist)`
- `(rest myslist)`
- `(student-mid (first (rest myslist)))`

The template for a list of students

We first break up a list of students using selectors for lists.

```
;; studentlist-template: (listof Student) → Any
(define (studentlist-template slist)
  (cond
    [(empty? slist) ...]
    [else (... (first slist)... (studentlist-template (rest slist))...)]))
```

Since (first slist) is a student, we can refine the template using the selectors for student.

```
;; studentlist-template: (listof Student) → Any
(define (studentlist-template slist)
  (cond
    [(empty? slist) ...]
    [else (... (student-name (first slist)) ...
               (student-assts (first slist)) ...
               (student-mid (first slist)) ...
               (student-final (first slist)) ...
               (studentlist-template (rest slist))...)]))
```

It may be convenient to define constants for sample data, as we did for structures.

This can reduce typing and make our examples and tests clearer.

Remember not to cut and paste from the Interactions window to the Definitions window.

```
;; Sample data
(define vw (make-student "Virginia Woolf" 100 100 100))
(define at (make-student "Alan Turing" 90 80 40))
(define an (make-student "Anonymous" 30 55 10))

(define classlist (cons vw (cons at (cons an empty))))
```

The function name-list

```
;; (name-list slist) produces a list of names in slist.
;; name-list: (listof Student) → (listof Str)
(check-expect (name-list classlist)
   (cons "Virginia Woolf" (cons "Alan Turing"
      (cons "Anonymous" empty))))

(define (name-list slist)
   (cond [(empty? slist) empty]
         [else (cons (student-name (first slist))
                     (name-list (rest slist)))]))
```

Computing final grades

Suppose we wish to determine final grades for students based on their marks in each course component (20% for assignments, 30% for the midterm, and 50% for the final exam).

How should we store the information we produce?

```
(define-struct grade (name mark))
;; A Grade is a (make-grade Str Num),
;; requires:
;;      0 <= mark <= 100
```

The function compute-grades

```
;; (compute-grades slist) produces a grade list from slist.
;; compute-grades: (listof Student) → (listof Grade)
;; Examples:
(check-expect (compute-grades empty) empty)
(check-expect (compute-grades classlist)
   (cons (make-grade "Virginia Woolf" 100)
      (cons (make-grade "Alan Turing" 62)
         (cons (make-grade "Anonymous" 27.5) empty))))
```

To enhance readability, we may choose to put the structure selectors in a helper function instead of in the main function.

```
(define (student-template s)
    (... (student-name s) ... (student-assts s) ...
         (student-mid s) ... (student-final s) ...))
(define (studentlist-template slist)
  (cond
    [(empty? slist) ...]
    [else (... (student-template (first slist)) ...
               (studentlist-template (rest slist))...)]))
```

Again, abstract list functions can be used instead if preferred.

The helper function final-grade

```
;; Constants for use in final-grade
(define assts-weight .20)
(define mid-weight .30)
(define final-weight .50)

;; (final-grade astudent) produces a grade from the astudent, with
;;    20 for assignments, 30 for midterm, and 50 for final
;; final-grade: Student → Grade
;; example:
(check-expect (final-grade vw) (make-grade "Virginia Woolf" 100))
```

```
(define (final-grade astudent)
  (make-grade
   (student-name astudent)
   (+ (* assts-weight (student-assts astudent))
      (* mid-weight (student-mid astudent))
      (* final-weight (student-final astudent)))))
(define (compute-grades slist)
  (map final-grade slist))
```

Condensed trace of compute-grades

```
(compute-grades myslist)
⇒ (compute-grades
   (cons (make-student "Virginia Woolf" 100 100 100)
      (cons (make-student "Alan Turing" 90 80 40)
         (cons (make-student "Anonymous" 30 55 10) empty))))
⇒ (cons (make-grade "Virginia Woolf" 100)
   (compute-grades
      (cons (make-student "Alan Turing" 90 80 40)
         (cons (make-student "Anonymous" 30 55 10) empty))))
```

```
⇒ (cons (make-grade "Virginia Woolf" 100)
    (cons (make-grade "Alan Turing" 62)
      (compute-grades
        (cons (make-student "Anonymous" 30 55 10) empty))))
⇒ (cons (make-grade "Virginia Woolf" 100)
    (cons (make-grade "Alan Turing" 62)
      (cons (make-grade "Anonymous" 27.5)
        (compute-grades empty))))
```

```
⇒ (cons (make-grade "Virginia Woolf" 100)
    (cons (make-grade "Alan Turing" 62)
      (cons (make-grade "Anonymous" 27.5) empty)))
```

Goals of this module

You should be comfortable with these terms: structure, field, constructor, selector, type predicate, dynamic typing, static typing, data definition, structure definition, template.

You should be able to write functions that consume and produce structures, including Posns.

You should be able to create structure and data definitions for a new structure, determining an appropriate type for each field.

You should know what functions are defined by a structure definition, and how to use them.

You should be able to write the template associated with a structure definition, and to expand it into the body of a particular function that consumes that type of structure.

You should understand the use of type predicates and be able to write code that handles mixed data.

You should understand how to process lists of structures.