Data Definition Guidelines : Structures

Example: a structure for capturing the name, weight, and favorite food for a boa

Data definition format:

```
;; A boa is a
;; (make-boa symbol number symbol)
(define-struct boa (name weight food))
```

Template format: the template pulls out all the fields from the structure

```
(define (fun-for-boa a-boa)
    (boa-name a-boa) ...
    (boa-weight a-boa) ...
    (boa-food a-boa) ... )
```

Data Definition Guidelines : Mixed Data

Example: a data definition for capturing various animals (boas, dillos, etc)

Data definition format:

```
;; An animal is
;;    - a boa, or
;;    - a dillo
```

Note that no `define-structs` are needed because we are only naming a set of possible animals. We are not introducing any new kinds of structures (assuming that boas and dillos are already defined).

Template format: the template does two things:

- decides which case of data we have
- pulls out any pieces of each kind of data

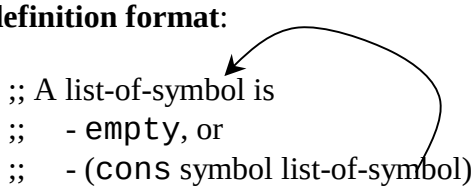
```
(define (fun-for-animal an-ani)
   (cond [(boa? an-ani)  ...
          (boa-name an-ani) ...
          (boa-weight an-ani) ...
          (boa-food an-ani) ... ]
         [(dillo? an-ani) ...
          (dillo-weight an-ani) ...
          (dillo-dead? an-ani) ... ]))
```

Note how this builds on templates for structures: within each case, if the case captures a structure, we pull out the same information as in the template for that structure.

Data Definition Guidelines : Lists of Atomic Data

Example: a list of symbols

Data definition format:

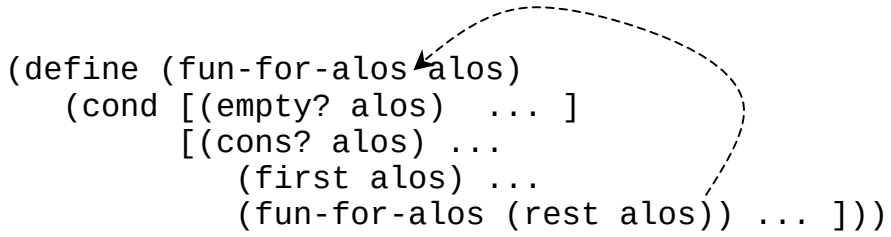


```
;; A list-of-symbol is
;;   - empty, or
;;   - (cons symbol list-of-symbol)
```

This definition has a similar format to mixed data definition. The main difference is that lists are recursive: they refer to themselves in the `cons` case. We annotate the data definition with the arrow to capture this self-reference.

Template format: the template does three things:

- decides which case of data we have
- pulls out pieces in the cons case (the case that has pieces)
- uses a recursive call to mimic the arrow in text



```
(define (fun-for-alos alos)
   (cond [(empty? alos)  ... ]
         [(cons? alos) ...
            (first alos) ...
            (fun-for-alos (rest alos)) ... ]))
```

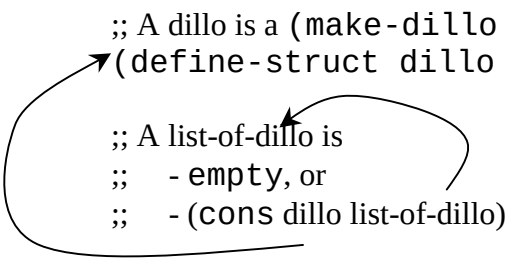
Note how similar this is to the template for mixed data: all we've really added is the arrow/recursive call, which is the only thing we added to the data definition. The dashed arrow in the template matches the arrow in the data defintion. **The number of arrows in the data definition and template must always match**!

.

Data Definition Guidelines : Lists of Structures (or any non-atomic data)

Example: a list of dillos

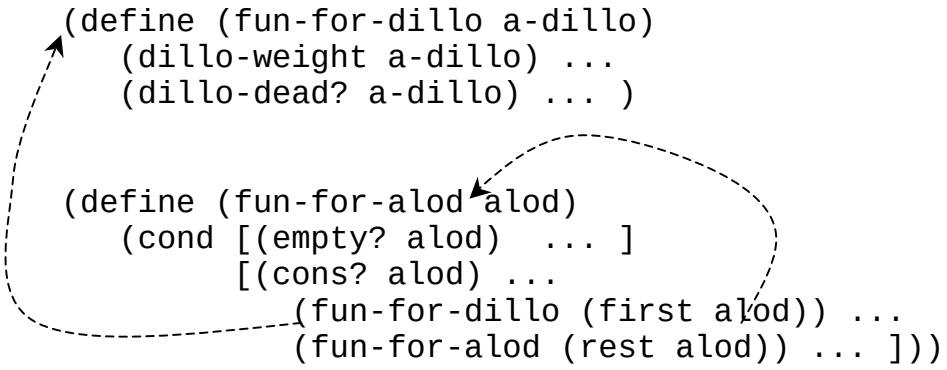
Data definition format:



This definition has a similar format to that for lists of atomic data. All we've added is an arrow from the element of the list (dillo) to the data definition for the dillo. We do this because the list contains items of a type of data that we defined (as opposed to one that is built-in). As before, the arrows capture references between data definitions.

Template format: the template contains **one function template per data definition**. Each

- decides which case of data we have
- pulls out pieces in the cons case (the case that has pieces)
- uses calls to template functions to mimic each arrow in text



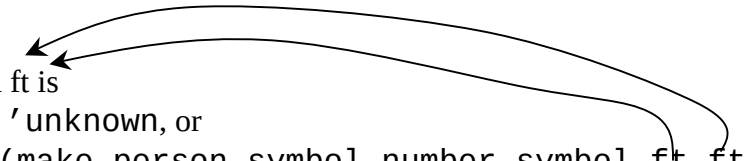
Note we've taken a template for structures (`fun-for-dillo`) and a template for lists (`fun-for-alod`) and simply connected them with an arrow that mimics the one in the data definition. Except for the new arrow, we've made no changes to the templates for those two data definitions.

Remember: **the number of arrows in the data definition and template must always match**!

Data Definition Guidelines : Trees

Example: a family tree with name, birth year, eye color, and parents for each person.

Data definition format:



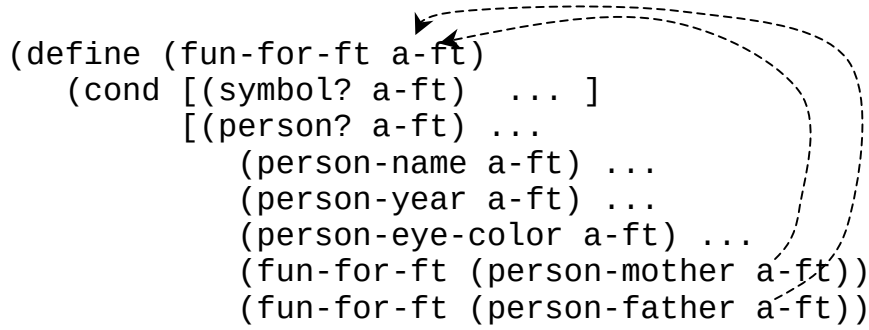
```
;; A ft is
;;  - 'unknown, or
;;  - (make-person symbol number symbol ft ft)
(define-struct person (name year eye-color mother father))
```

This definition has a similar format to that for lists, but there are two differences:

- We have two recursive arrows instead of one (since father and mother are both family trees)
- We don't use `cons`, since we're not building lists.
- We use `'unknown` rather than `empty`, since we're not building lists

Template format: as before, the template

- decides which case of data we have
- pulls out pieces in appropriate cases
- uses recursive calls to mimic each arrow



```
(define (fun-for-ft a-ft)
   (cond [(symbol? a-ft)  ... ]
         [(person? a-ft) ...
            (person-name a-ft) ...
            (person-year a-ft) ...
            (person-eye-color a-ft) ...
            (fun-for-ft (person-mother a-ft)) ...
            (fun-for-ft (person-father a-ft)) ... ]))
```

Note that the template doesn't use `cons`, since there is no `cons` in the data definition. Otherwise, this uses all of the same principles we've used to develop previous templates.