Module 10: Local and functional abstraction

Readings: HtDP, Intermezzo 3 (Section 18); Sections 19-23.

We move to the Intermediate teaching language with the introduction of local definitions.

We will cover material on functional abstraction in a somewhat different order than the text.

CS 115 will cover built-in functions that consume functions as inputs.

CS 116 will cover functions that produce functions, and how to write functions that produce functions.

Local definitions

The functions and special forms we've seen so far can be arbitrarily nested – except define and check-expect.

So far, definitions have to be made "at the top level", outside any expression.

The Intermediate language provides the special form local, which contains a series of local definitions plus an expression using them, of the form

(local (def1 . . . defn) exp)

What use is this?

Motivating local definitions

Recall the function swap-parts from Module 2.

The function used three helper functions.

```
(define (mid num)
  (quotient num 2))
(define (front-part mystring)
  (substring mystring 0 (mid (string-length mystring))))
(define (back-part mystring)
  (substring mystring (mid (string-length mystring))
                      (string-length mystring)))
```

The helper function mid is a helper function of the helper functions front-part and back-part.

```
(define (swap-parts mystring)
  (string-append (back-part mystring) (front-part mystring)))
```

Our solution was perfectly acceptable.

However, repeated applications, such as

(mid (string-length mystring)),

make it a bit hard to read.

It would be nice to replace repeated applications by a constant.

The special form local allows us to define a constant or a function within another function.

```
(define (swap-parts mystring)
  (local
    ((define mid (quotient (string-length mystring) 2))
     (define front (substring mystring 0 mid))
     (define back (substring mystring mid
                             (string-length mystring))))
    (string-append back front)))
```

Like cond, local results in double parentheses.

Optional: use square brackets to improve readability.

```
(define (swap-parts mystring)
  (local
    [(define mid (quotient (string-length mystring) 2))
     (define front (substring mystring 0 mid))
     (define back (substring mystring mid (string-length mystring)))]
    (string-append back front)))
```

Using local to reuse names

We have reused names before:

- n is bound to a value by define
- n is the name of a parameter

```
(define n 10)
(define (myfun n) (+ 2 n))
(myfun 6)
```

The function application (myfun 6) produces 8, not 12, due to the substitution rules for function application.

A define within a local expression may rebind a name that has already been bound to another value or expression.

```
(define (my-fun n)
  (local
    [(define (local-fun n)
       (* n 10))]
    (+ n (local-fun n))))
```

The substitution rules for local must handle this.

The semantics of local

Key ideas:

- Create a new, unique name for each identifier defined in a local definition (e.g. mid becomes mid_0).
- Bind the new name to the value.
- Substitute the new name for the old name everywhere in the expression.
- Move all of the local definitions to the top level, evaluate, and continue.

Evaluating swap-parts

```
(swap-parts "aside")
⇒ ( local [(define mid (quotient (string-length "aside") 2))
           (define front (substring "aside" 0 mid))
           (define back (substring "aside" mid
                                   (string-length "aside")))]
       (string-append back front))
 ⇒ (define mid_0 (quotient (string-length "aside") 2))
   (define front_0 (substring "aside" 0 mid_0))
   (define back_0 (substring "aside" mid_0 (string-length "aside")))
   (string-append back_0 front_0)
```

```
⇒ (define mid_0 (quotient 5 2))
  (define front_0 (substring "aside" 0 mid_0))
  (define back_0 (substring "aside" mid_0 (string-length "aside")))
  (string-append back_0 front_0)
⇒ (define mid_0 2)
  (define front_0 (substring "aside" 0 mid_0))
  (define back_0 (substring "aside" mid_0 (string-length "aside")))
  (string-append back_0 front_0)
```

```
⇒ (define mid_0 2)
  (define front_0 (substring "aside" 0 2)
  (define back_0 (substring "aside" mid_0 (string-length "aside")))
  (string-append back_0 front_0)
⇒ (define mid_0 2)
  (define front_0 "as")
  (define back_0 (substring "aside" mid_0 (string-length "aside")))
  (string-append back_0 front_0)
```

```
⇒ (define mid_0 2)
  (define front_0 "as")
  (define back_0 (substring "aside" 2 (string-length "aside")))
  (string-append back_0 front_0)
⇒ (define mid_0 2)
  (define front_0 "as")
  (define back_0 (substring "aside" 2 5))
  (string-append back_0 front_0)
```

```
⇒ (define mid_0 2)
  (define front_0 "as")
  (define back_0 "ide")
  (string-append back_0 front_0)
⇒ (define mid_0 2)
  (define front_0 "as")
  (define back_0 "ide")
  (string-append "ide" front_0)
```

⇒ (define mid_0 2)

(define front_0 "as")

(define back_0 "ide")

(string-append "ide" "as")

⇒ "ideas"

Substitution rule

An expression of the form

(local [(define x1 exp1) ... (define xn expn)] bodyexp) is handled as follows:

x1 is replaced with a **fresh** identifier (call it x1_0) everywhere in exp1 through expn and bodyexp.

x2 is replaced with x2_0 everywhere in exp1 through expn and bodyexp.

This process is repeated with x3 through xn.

Then, all the definitions are lifted out to the top level, yielding

```
(define x1_0 exp1)
(define x2_0 exp2)
...
(define xn_0 expn)
bodyexp
```

where the expressions have been modified to use the new names.

Read Intermezzo 3 (Section 18).

Nested local expressions

It isn't always possible to define local at the beginning of the function definition, because the definition might make assumptions that are only true in part of the code.

A typical example is that of using a list function, like first or rest, which must consume a non-empty list.

When there is one local definition that can be used throughout and one not, we end up with nested local expressions.

Assume preprocess and core-fn each consume a list and both have been defined.

```
(local
  [(define refined-list (preprocess alist))]
  (cond
    [(empty? refined-list) . . . ]
    [else
      (local
        [(define core-list (core-fn refined-list))]
        ;; code using core-list goes here
        )]))
```

Using local for common subexpressions

A subexpression used twice within a function body always yields the same value.

Using local to give the reused subexpression a name improves the readability of the code.

In the following example, the function eat-apples removes all occurrences of the symbol 'apple from a list of symbols.

The subexpression (eat-apples (rest alist)) occurs twice in the code.

The function eat-apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
      (cond
        [(not (symbol=? (first alist) 'apple))
         (cons (first alist) (eat-apples (rest alist)))]
        [else
         (eat-apples (rest alist))])]))
```

Using local in eat-apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
      (local [(define ate-rest (eat-apples (rest alist)))]
        (cond
          [(not (symbol=? (first alist) 'apple))
           (cons (first alist) ate-rest)]
          [else ate-rest]))]))
```

In the function eat-apples, the subexpression

(eat-apples (rest alist))

appears in two different answers of the same cond expression, so only one of them will ever be evaluated.

In the next example, the subexpression

(list-max (rest alon))

appears twice. The first appearance is always evaluated, and sometimes both are. In this case, using local is more efficient, as well as more readable.

The function list-max

```
;; list-max: (listof num)[nonempty] → num
;; Produces maximum of list of numbers alon.
(define (list-max alon)
  (cond
    [(empty? (rest alon))  (first alon)]
    [else
     (cond
       [(> (first alon) (list-max (rest alon)))  (first alon)]
       [else  (list-max (rest alon))])]))
```

Using local in list-max

```
;; list-max2: (listof num)[nonempty] → num
;; Produces maximum of list of numbers (2nd version).
(define (list-max2 alon)
  (cond
    [(empty? (rest alon))  (first alon)]
    [else
     (local [(define max-rest (list-max2 (rest alon)))]
       (cond
         [(> (first alon) max-rest)  (first alon)]
         [else  max-rest]))]))
```

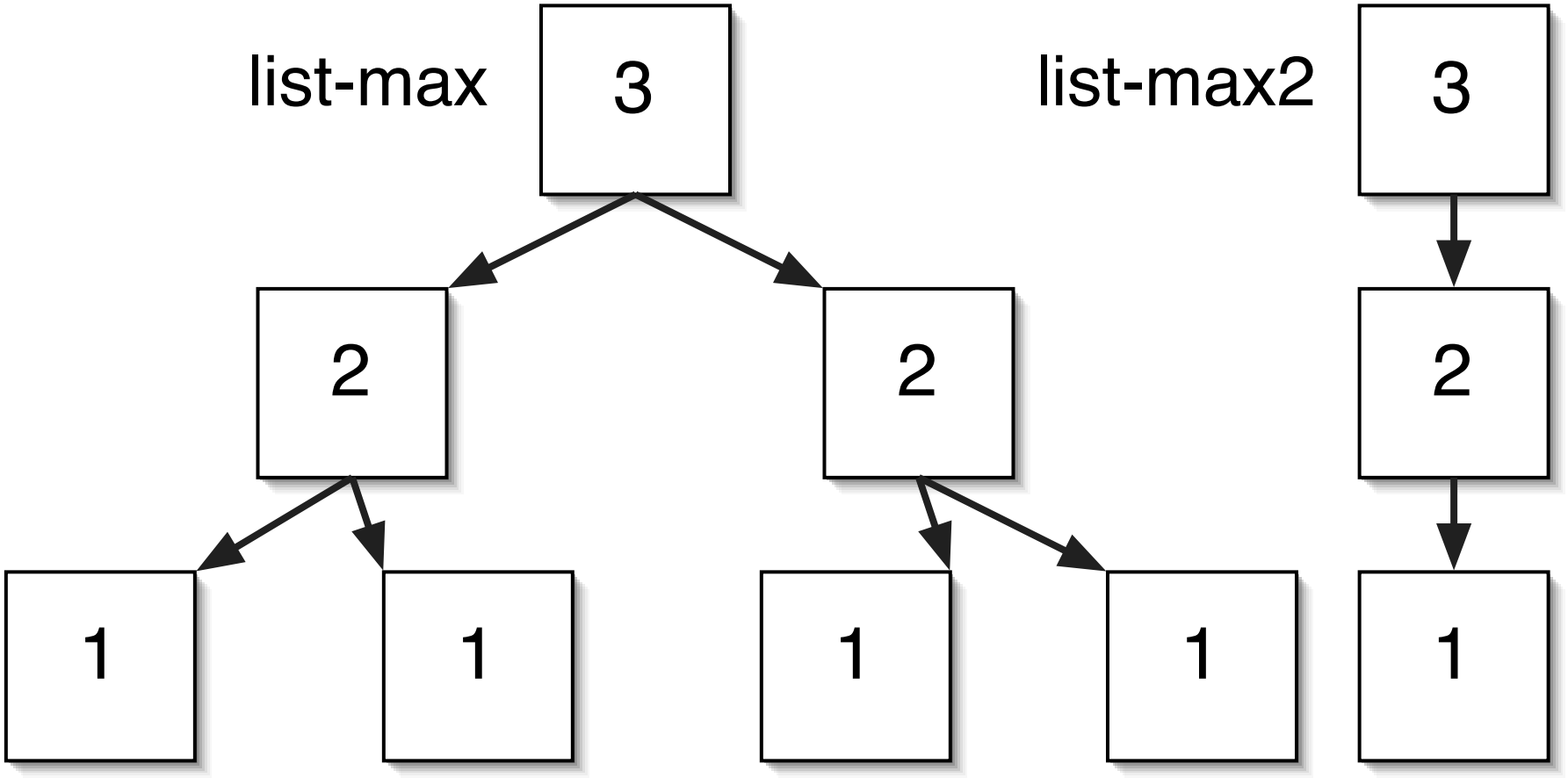
Using local can improve efficiency

You might expect that the first version does no more than twice the work of the second version.

But the first version may make two recursive calls, and each of them may make two, and so on.

If we run each version on an increasing list of three numbers, and draw a box for each call containing the length of its argument, we can see how the work adds up.

Tracing versions of list-max



Using local to improve ancestors

```
(define (ancestors t tname)
    ...
    [(t-ancient? t)
     (cond
       [(equal? (t-ancient-name t) tname)  (list tname)]
       [(cons? (ancestors (t-ancient-left t) tname))
        (cons (t-ancient-name t) (ancestors (t-ancient-left t) tname))]
       [(cons? (ancestors (t-ancient-right t) tname))
        (cons (t-ancient-name t) (ancestors (t-ancient-right t) tname))]
       [else  false])]))
```

```
(define (ancestors t tname)
  ...
  [(t-ancient? t)
   (local
     [(define left-result (ancestors (t-ancient-left t) tname))
      (define right-result (ancestors (t-ancient-right t) tname))]
     (cond
       [(equal? (t-ancient-name t) tname) (list tname)]
       [(cons? left-result) (cons (t-ancient-name t) left-result)]
       [(cons? right-result) (cons (t-ancient-name t) right-result)]
       [else false]))]))
```

Using local for smaller tasks

Sometimes we choose to use local in order to name subexpressions mnemonically to make the code more readable, even if they are not reused.

This may make the code longer.

Recall our function to compute the distance between two points.

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

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

(define (distance posn1 posn2)
  (local [(define delta-x (- (posn-x posn1) (posn-x posn2)))
          (define delta-y (- (posn-y posn1) (posn-y posn2)))
          (define sqrdx (sqr delta-x))
          (define sqrdy (sqr delta-y))
          (define sqrsum (+ sqrdx sqrdy))]
    (sqrt sqrsum)))
```

Using local for encapsulation

Encapsulation is the process of grouping things together in a "capsule".

We have already seen data encapsulation in the use of structures.

Encapsulation can also be used to hide information. Here the local bindings are not visible (have no effect) outside the local expression.

We can bind names to functions as well as values in a local definition.

Evaluating the local expression creates new, unique names for the functions just as for the values.

This is known as **behaviour** encapsulation.

Behaviour encapsulation allows us to move helper functions within the function that uses them, so they are invisible outside the function.

```
(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)
  (cond [(empty? slist) empty]
        [else (cons (final-grade (first slist))
                    (compute-grades (rest slist)))]))
```

```
(define (compute-grades slist)
  (local
    [(define (final-grade astudent)
       (make-grade
        (student-name astudent)
        (+ (* assts-weight (student-assts astudent))
           (* mid-weight (student-mid astudent))
           (* final-weight (student-final astudent)))))]
    (cond [(empty? slist) empty]
          [else (cons (final-grade (first slist))
                      (compute-grades (rest slist)))])))
```

```
(define (my-sort alon)
  (cond
    [(empty? alon) empty]
    [ else (insert (first alon) (my-sort (rest alon)))]))

(define (insert n alon)
  (cond
    [(empty? alon) (cons n empty)]
    [ else (cond
             [(>= n (first alon)) (cons n alon)]
             [ else (cons (first alon) (insert n (rest alon)))])]))
```

```
(define (my-sort alon)
  (local [(define (insert n alon)
            (cond
              [(empty? alon) (cons n empty)]
              [ else (cond
                [(>= n (first alon)) (cons n alon)]
                [ else (cons (first alon) (insert n (rest alon)))])]))]
    (cond
      [(empty? alon) empty]
      [ else (insert (first alon) (my-sort (rest alon)))])))
```

Note that a full design recipe is not needed for local helper functions on assignment submissions.

You should still develop examples and tests for helper functions, and test them outside local expressions if possible.

Once you have faith that they work, you can move them into a local expression, and delete the examples and tests.

A contract and purpose are still required (omitted in these slides for space reasons).

What is abstraction?

Abstraction consists of

- finding similarities or common aspects, and
- forgetting unimportant differences.

For a single function, differences in parameter values are forgotten, and the similarity is captured in the function body.

For multiple functions, similarity is captured in templates.

For multiple functions, further abstraction is possible.

Eating apples

```
(define (eat-apples alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
      (cond
        [(not (symbol=? (first alist) 'apple))
         (cons (first alist) (eat-apples (rest alist)))]
        [else (eat-apples (rest alist))])]))
```

Selecting even numbers

```
(define (select-even alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
      (cond
        [(even? (first alist)) (cons (first alist) (select-even (rest alist)))]
        [else (select-even (rest alist))])]))
```

Abstracting from these examples

Similarity: general structure (removing certain items)

Difference: predicate used to decide what to remove

Goal: form an **abstract list function** that consumes the predicate.

Functions are *first-class* values in the Intermediate Student Language.

First-class values can be bound to constants, put in lists and structures, consumed as arguments, and produced as results.

The abstract list function filter

```
(define (filter pred alist)
  (cond
    [(empty? alist) empty]
    [(cons? alist)
      (cond
        [(pred (first alist))
          (cons (first alist) (filter pred (rest alist)))]
        [else
          (filter pred (rest alist))])]))
```

We can simplify the code somewhat.

```
(define (filter pred alist)
  (cond
    [(empty? alist) empty]
    [(pred (first alist))
             (cons (first alist) (filter pred (rest alist)))]
    [else (filter pred (rest alist))]))
```

Tracing filter

(filter even? (list 6 7 8))

⇒ (cons 6 (filter even? (list 7 8)))

⇒ (cons 6 (filter even? (list 8)))

⇒ (cons 6 (cons 8 (filter even? empty)))

⇒ (cons 6 (cons 8 empty))

The **abstract list function** filter performs the general operation of selecting items from lists.

Scheme provides such functions to apply common patterns of structural recursion.

Using filter

```
(define (select-even alist) (filter even? alist))
(define (symbol-not-apple? item) (not (symbol=? item 'apple)))
(define (eat-apples alist) (filter symbol-not-apple? alist))
```

The built-in function filter consumes a predicate specifying which elements of the list are to be kept.

The predicate must be a one-parameter function producing a boolean, where the type of the parameter is same as the type of the elements of the list.

Advantages of functional abstraction

Functional abstraction is the process of creating abstract functions such as filter.

It reduces code size.

It avoids cut-and-paste.

Bugs can be fixed in one place instead of many.

Improving one functional abstraction improves many applications.

Additions to syntax and semantics with Intermediate

An expression can now be a primitive operation or a function.

Names of functions are now values.

In the substitution rules, if the first position in an application is an expression, it must be evaluated along with the other arguments.

Abstracting from examples

```
(define (negate-list numlist)
  (cond
    [(empty? numlist) empty]
    [else (cons (— (first numlist)) (negate-list (rest numlist)))]

(define (compute-grades rlist)
  (cond
    [(empty? rlist) empty]
    [else (cons (final-grade (first rlist))
                (compute-grades (rest rlist)))]))
```

The abstract list function map

```
(define (map f alist)
  (cond
    [(empty? alist) empty]
    [else (cons (f (first alist))
                (map f (rest alist)))]))
```

For this and other built-in abstract list functions, see the table on page 313 of the text (Figure 57 in Section 21.2).

Tracing map

(map sqr (list 3 6 5))

⇒ (cons (sqr 3) (map sqr (list 6 5)))

⇒ (cons 9 (map sqr (list 6 5)))

⇒ (cons 9 (cons (sqr 6) (map sqr (list 5))))

⇒ (cons 9 (cons 36 (map sqr (list 5))))

⇒ (cons 9 (cons 36 (cons (sqr 5) (map sqr empty))))

⇒ (cons 9 (cons 36 (cons 25 (map sqr empty))))

⇒ (cons 9 (cons 36 (cons 25 empty)))

The abstract list function map performs the operation of transforming a list element-by-element into another list of the same length.

(map f (list x1 x2 . . . xn)) has the same effect as
(list (f x1) (f x2) . . . (f xn)).

Short definitions using map:

```
(define (negate-list numlist) (map - numlist))
(define (compute-grades rlist) (map final-grade rlist))
```

The function consumed by map must be a one-parameter function where the type of the parameter is the same as the type of the elements of the list.

```
(define (product-of-numbers alist)
  (cond
    [(empty? alist) 1]
    [else (* (first alist) (product-of-numbers (rest alist)))]))

(define (my-length alist)
  (cond
    [(empty? alist) 0]
    [else (+ 1 (my-length (rest alist)))]))
```

```
(define (my-list-fun alist)
  (cond
    [(empty? alist) ...]
    [else ... (first alist) ... (my-list-fun (rest alist)) ...]))
```

To fill in the template:

Replace the first ellipsis by a base value.

Combine (first alist) and the result of a recursive call on (rest alist).

Parameters for the abstract list function: base value and combining function.

The abstract list function foldr

```
(define (foldr combine base alist)
  (cond
    [(empty? alist) base]
    [else (combine
           (first alist)
           (foldr combine base (rest alist)))]))
```

foldr is also a built-in function in Scheme.

Tracing foldr

(foldr f 0 (list 3 6 5))

$\Rightarrow$ (f 3 (foldr f 0 (list 6 5)))

$\Rightarrow$ (f 3 (f 6 (foldr f 0 (list 5)))

$\Rightarrow$ (f 3 (f 6 (f 5 (foldr f 0 empty)))

$\Rightarrow$ (f 3 (f 6 (f 5 0))) $\Rightarrow$ . . .

Intuitively, the effect of the application

(foldr f b (list x1 x2 . . . xn)) is to compute the value of the expression

(f x1 (f x2 (. . . (f xn b) . . .))).

foldr is short for “fold right”.

It can be viewed as “folding” a list using the provided combine function, starting from the right-hand end of the list.

It can be used to implement map, filter, and other abstract list functions.

Using foldr

```
(define (product-of-numbers alist) (foldr * 1 alist))
```

If alist is (list x1 x2 ... xn), then by our intuitive explanation of foldr, the expression (foldr * 1 alist) reduces to

```
(* x1 (* x2 (* ... (* xn 1) ...)))
```

Thus foldr does all the work of the template for processing lists in the case of product-of-numbers.

The combine function provided to foldr consumes two parameters:

- an item in the list that foldr consumes and
- the result of applying foldr to the rest of the list.

In product-of-numbers the function * multiplies an element with the product of the rest of the list.

How does my-length use these two parameters?

For my-length, the first argument to the function contributes 1 to the count; its actual value is irrelevant.

Thus the function provided to foldr in this case can ignore the value of the first parameter, and just add 1 to the result on the rest of the list.

```
(define (my-length alist)
  (local
      [(define (inc x y) (add1 y))]
    (foldr inc 0 alist)))
```

Using foldr to produce lists

The functions we provide to foldr can also produce cons expressions, since these are also values.

Example: using foldr for negate-list.

How do we produce the negate list from

- an item in the list and
- the result of the recursive call on the rest of the list?

neg-combine takes the element, negates it, and conses it onto the result of the recursive call.

```
(define (neg-combine item result-on-rest)
  (cons (- item) result-on-rest))


(define (negate-list alist)
  (local
      ;; neg-combine: num (listof num) → (listof num)
      ;; Produces the cons of negative item and
      ;; result-on-rest.
      [(define (neg-combine item result-on-rest)
         (cons (- item) result-on-rest))]
    (foldr neg-combine empty alist)))
```

Defining map using foldr

```
(define (map f alist)
  (cond
    [(empty? alist) empty]
    [else (cons (f (first alist))
                (map f (rest alist)))]))
```

What are the base value and the combine function?

The base value is empty.

The combine function applies f to its first argument, then cons the result onto the recursive call on the rest of the list.

```
(define (map f alist)
  (local
      [(define (map-combine item result-on-rest)
         (cons (f item) result-on-rest))]
    (foldr map-combine empty alist)))
```

Boolean functions and foldr

To check whether a predicate p produces true for every element in a list alist, we might be tempted to try:

```
(foldr and true (map p alist))
```

Problem: and is not a function, but a special form, and this produces an error.

Solution: Scheme provides andmap, which can be used like this:

```
(andmap p alist)
```

For the same reason, ormap is provided.

Imperative languages, which tend to provide inadequate support for recursion, usually provide looping constructs such as “while” and “for” to perform repetitive actions on data.

Abstract list functions cover many of the common uses of such looping constructs.

Our implementation of these functions is not difficult to understand, and we can write more if needed, but the set of looping constructs in a conventional language is fixed.

Anything that can be done with the list template can be done using foldr, without explicit recursion.

Does that mean that the list template is obsolete?

No. Experienced Scheme programmers still use the list template, for reasons of readability and maintainability.

Abstract list functions should be used judiciously, to replace relatively simple uses of recursion.

Goals of this module

You should understand the syntax, informal semantics, and formal substitution semantics for the local special form.

You should be able to use local to avoid repetition of common subexpressions, to improve readability of expressions, and to improve efficiency of code.

You should understand the idea of encapsulation of local helper functions.

You should be able to match the use of any constant or function name in a program to the binding to which it refers.

You should understand the idea of functions as first-class values, and how they can be supplied as arguments.

You should be familiar with map, filter, and foldr, understand how they abstract common recursive patterns, and be able to use them to write code.

You should understand how to do step-by-step evaluation of programs written in the Intermediate language that make use of functions as values.

Summing up CS 115

With only a few language constructs we have described and implemented ideas from introductory computer science in a brief and natural manner.

We have done so without many of the features (static types, mutation, I/O) that courses using conventional languages have to introduce on the first day. The ideas we have covered carry over into languages in more widespread use.

We hope you have been convinced that the goal of computer science is to implement useful computation in a way that is correct and efficient as far as the machine is concerned, but that is understandable and extendable as far as other humans are concerned.

These themes will continue in CS 116, but new themes will be added, and a new programming language using a different paradigm will be studied (though we will continue to use Scheme as well).

Looking ahead to CS 116 and beyond

We have been fortunate to work with very small languages (the teaching languages) writing very small programs which operate on small amounts of data.

In CS 116, we will broaden our scope, moving towards the messy but also rewarding realm of the real world.

In subsequent courses we will address such questions as:

- How do we organize a program that is bigger than a few screenfuls, or large amounts of data?
- How do we reuse and share code, apart from cutting-and-pasting it into a new program file?
- How do we design programs so that they run efficiently?

These are issues which arise not just for computer scientists, but for anyone making use of computation in a working environment.

We can build on what we have learned this term in order to meet these challenges with confidence.