# CS 9D Study Guide

© Copyright 2005, University of California

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Structure of quizzes and programs in CS 9D.</td>
<td>2</td>
</tr>
<tr>
<td>Orientation</td>
<td>3</td>
</tr>
<tr>
<td>Program—Recursion exercises</td>
<td>4</td>
</tr>
<tr>
<td>Quiz—Functions and recursion</td>
<td>9</td>
</tr>
<tr>
<td>Program—Twenty-One</td>
<td>11</td>
</tr>
<tr>
<td>Quiz—Higher-order functions.</td>
<td>16</td>
</tr>
<tr>
<td>Program—The Doctor program</td>
<td>19</td>
</tr>
<tr>
<td>Quiz—Evaluation with lists</td>
<td>24</td>
</tr>
<tr>
<td>Quiz—List-processing recursion</td>
<td>27</td>
</tr>
<tr>
<td>Program—Type inference</td>
<td>30</td>
</tr>
<tr>
<td>Program—Font design</td>
<td>33</td>
</tr>
<tr>
<td>Quiz—Higher-order functions with lists</td>
<td>42</td>
</tr>
<tr>
<td>What’s functional programming all about?</td>
<td>45</td>
</tr>
<tr>
<td>Using script</td>
<td>46</td>
</tr>
<tr>
<td>Sequential programming in Scheme</td>
<td>47</td>
</tr>
<tr>
<td>Turtle graphics primitive procedures.</td>
<td>48</td>
</tr>
<tr>
<td>Emacs reference</td>
<td>51</td>
</tr>
<tr>
<td>Notes from tutoring sessions</td>
<td>56</td>
</tr>
</tbody>
</table>
Introduction

In CS 9D, you learn to program in Scheme, a dialect of Lisp. You are assumed to know how to program already in a high-level language such as Pascal, Fortran, C, C++, or Java. Programming in this course will differ from what you’re used to, however; you’ll be working in the functional style, that is, without the side effects of assignment statements, but with the power of higher-order functions that take functions as arguments or produces a function as a result.

Course material consists of quizzes, which test your knowledge of language and low-level conceptual details, and programming assignments, which exercise your overall command of the language. This volume supplies a framework for the course. It contains the following:

Study guides. Each study guide focuses on a particular programming topic. It provides references to textbook material describing the topic, and suggests exercises for self-study. The study guides reference other sections in this volume, along with the following text and documents.

- Concrete Abstractions, Max Hailperin et al. (Brooks/Cole, 1999).

Programming assignments. Each one has a header page (this tells you the title and related topics) that is followed by the actual assignment.

Sample quiz questions, with solutions. These help you prepare for the quizzes.

Comments on the textbooks

The two textbooks cover many of the same topics, with much the same emphasis. Concrete Abstractions provides more examples, is more explicit about problem-solving strategies, and assumes less mathematical preparation of the reader than Structure and Interpretation of Computer Programs. The latter text has long been used in CS 61A and is a good reference for students intending to take that course after CS 9D.
Structure of quizzes and programs in CS 9D

The following table outlines the relationship between quizzes and programs. All the material for a particular grouping must be completed before material in the next grouping; however, quizzes and programs within a group may be done in any order.

<table>
<thead>
<tr>
<th>group</th>
<th>programming assignments</th>
<th>quizzes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Orientation</td>
<td>Functions and recursion</td>
</tr>
<tr>
<td></td>
<td>Recursion exercises</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Twenty-One</td>
<td>Higher-order functions</td>
</tr>
<tr>
<td>c</td>
<td>The Doctor program</td>
<td>Evaluation with lists</td>
</tr>
<tr>
<td></td>
<td>Type inference</td>
<td>List-processing recursion</td>
</tr>
<tr>
<td>d</td>
<td>Font design</td>
<td>Higher-order functions with lists</td>
</tr>
</tbody>
</table>

Note that this breakdown is different from what’s required to satisfy course deadlines. For information about deadlines, consult the “Information and Regulations” document.
**Orientation**

**Goals**
In this activity, you familiarize yourself with software to compute deadline penalties in the self-paced courses and supply us with some administrative information that makes it easier to contact you.

**Readings**
The “Information and Regulations” pamphlet.

**Problem**
The “Orientation” assignment will be distributed at the Self-Paced Center. You must complete it to enable any of your other work to be recorded. Complete it as early in the semester as possible.
Program—Recursion exercises

Goals
Recursion and function calling are the primary mechanisms for control flow in Scheme. The exercises for this segment give you practice defining recursive functions.

Related quizzes
Functions and recursion.

Readings
Concrete Abstractions, chapters 1 through 4.
Structure and Interpretation of Computer Programs, sections 1.1 through 1.2.2.

Miscellaneous information
The terms procedure and function are used essentially as synonyms in this document. The two textbooks use different formats for defining Scheme functions. In Concrete Abstractions, for example, a function that returns the square of its argument is defined as

```
(define square
  (lambda (x) (* x x)) )
```

while in Structure and Interpretation of Computer Programs, the definition appears as

```
(define (square x)
  (* x x) )
```

The two formats are equivalent. Anecdotal evidence suggests that the second format helps novice programmers get more easily accustomed to Scheme evaluation, since a function is defined in the same way that it’s used. The first format, on the other hand, seems to help students later in the course when the lambda construct is used in more flexible ways. You may use either, but you should understand both.

If you’re using an account on the EECS instructional systems, stk is the Scheme interpreter.

Problem
Do both sets of exercises on the following pages.
Exercise set 1

Exercises 3.9, 3.10, 3.11, and 3.12 in *Concrete Abstractions* section 3.5. For grading, bring in test results for exercises 3.10 and 3.12 as well as for 3.11. Test your code on inputs that you can easily verify by hand.

**Miscellaneous requirements**

For this exercise and for all subsequent exercises, you should bring a listing of your Scheme code and a transcript of your tests to the Self-Paced Center for grading. (On a UNIX account, the `script` command, described in an appendix of this document, will produce such a transcript.) Your code should be indented to show its structure; each function should be commented that describes its arguments and result; function and parameter names should be suggestive of their purpose. Your tests should exercise all cases of any `cond` or `if` in your code.

Scheme has a counterpart to the assignment statement of other languages (it’s named `set!`), but its use is forbidden for the exercises in this course. A one-time assignment is done using the `define` or `let` operator; both are allowed in CS 9D.

**Checklist**

- All questions answered, with sufficient test results for 3.10, 3.11, and 3.12.
- A printed transcript of test results.
- A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.
- Reasonable names for functions and parameters.
- Avoidance of `set!`. 
Exercise set 2

The function below (adapted from code in *The Structure and Interpretation of Computer Programs*) counts the number of ways to form a given amount of money with pennies, nickels, dimes, quarters, and half-dollars.

```
(define (num-ways-to-form amount)
  (num-ways-to-form-with amount 5))

(define (num-ways-to-form-with amount num-denominations)
  (cond
    ((= amount 0) 1)
    ((< amount 0) 0)
    ((= num-denominations 0) 0)
    (else
     (+
      (num-ways-to-form-with ; first coin used
       (- amount (first-denomination num-denominations))
       num-denominations)
      (num-ways-to-form-with ; first coin not used
       amount
       (- num-denominations 1)))))

(define (first-denomination num-denominations)
  (cond
    ((= num-denominations 1) 1)
    ((= num-denominations 2) 5)
    ((= num-denominations 3) 10)
    ((= num-denominations 4) 25)
    ((= num-denominations 5) 50) ) )
```

For example, (num-ways-to-form 25) will return 13, representing the following ways to form 25 cents with pennies, nickels, dimes, quarters, and half-dollars:

- 0 pennies + 0 nickels + 0 dimes + 1 quarter
- 0 pennies + 1 nickel + 2 dimes + 0 quarters
- 0 pennies + 3 nickels + 1 dime + 0 quarters
- 0 pennies + 5 nickels + 0 dimes + 0 quarters
- 5 pennies + 0 nickels + 2 dimes + 0 quarters
- 5 pennies + 2 nickels + 1 dime + 0 quarters
- 5 pennies + 4 nickels + 0 dimes + 0 quarters
- 10 pennies + 1 nickel + 1 dime + 0 quarters
- 10 pennies + 3 nickels + 0 dimes + 0 quarters
- 15 pennies + 0 nickels + 1 dime + 0 quarters
- 15 pennies + 2 nickels + 0 dimes + 0 quarters
- 20 pennies + 1 nickel + 0 dimes + 0 quarters
- 25 pennies + 0 nickels + 0 dimes + 0 quarters

In the exercises of this set, you write functions that are similar to the above code.
Exercises

1. The partitions of a positive integer \( n \) are the different ways to break the integer into pieces. The number 5 has seven partitions:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(one piece)</td>
</tr>
<tr>
<td>4, 1</td>
<td>(two pieces)</td>
</tr>
<tr>
<td>3, 2</td>
<td>(two pieces)</td>
</tr>
<tr>
<td>3, 1, 1</td>
<td>(three pieces)</td>
</tr>
<tr>
<td>2, 2, 1</td>
<td>(three pieces)</td>
</tr>
<tr>
<td>2, 1, 1, 1</td>
<td>(four pieces)</td>
</tr>
<tr>
<td>1, 1, 1, 1, 1</td>
<td>(five pieces)</td>
</tr>
</tbody>
</table>

The order of the pieces doesn’t matter, so the partition 2, 3 is the same as the partition 3, 2 and thus isn’t counted twice. 0 has one partition.

Suppose we wanted to compute the number of partitions of 5. A CS 9D student proposes to do it by computing the following sum:

\[
\text{the number of partitions of 0} + \text{the number of partitions of 1} + \text{the number of partitions of 2} + \text{the number of partitions of 3} + \text{the number of partitions of 4}
\]

(i.e. the number of partitions of 5 that start with 5)

(the number of partitions of 5 that start with 4)

(the number of partitions of 5 that start with 3)

(the number of partitions of 5 that start with 2)

(the number of partitions of 5 that start with 1)

What’s wrong with this idea?
2. For reasons relating to your answer to exercise 1, a function that returns the number of partitions, none of whose pieces is bigger than a given size, will be useful. Here's how.

```scheme
(define (number-of-partitions n)
  ;; Return the number of partitions of the integer n.
  (cond
    ((< n 0) 0) ;; can't partition a negative number
    ((= n 0) 1) ;; one partition of zero
    (else (number-of-bounded-partitions n n))))
```

The function `number-of-bounded-partitions` is the one just mentioned. Here are the first few lines of its definition.

```scheme
(define (number-of-bounded-partitions n largest)
  ;; Return the number of partitions of the integer n, no piece of which is greater than largest.
  (cond
    ((= largest 0) 0) ;; none with largest piece = 0
    ((< n 0) 0) ;; as above
    ((= n 0) 1) ;; as above
    ((< n largest) (number-of-bounded-partitions n n)); to save some time
    (else _______________________________________________ )
  )
)
```

Complete the definition of the function `number-of-bounded-partitions`, and test your pair of functions with values of `n` ranging from 1 to 5.

3. Compare the function `number-of-partitions` with the `num-ways-to-form` function by filling in the blank in the following statement.

   Counting partitions is like counting ways to form a given amount with coins of the following denominations:
   __________________________________________________.

4. Write and test a function to count the ways an integer `n` can be expressed as the sum of different positive integers. Examples:

   7 can be expressed as the sum of different positive integers in five ways:
   7, 6+1, 5+2, 4+3, and 4+2+1.

   9 can be expressed as the sum of different positive integers in eight ways:
   9, 8+1, 7+2, 6+3, 6+2+1, 5+4, 5+3+1, and 4+3+2.

**Checklist**

- All questions answered, with test results as specified for exercises 2 and 4.
- A printed transcript of test results.
- A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.
- Reasonable names for functions and parameters.
- Avoidance of `set!`. 
Quiz—Functions and recursion

This quiz gives you practice with the definition and analysis of functions, mostly recursive functions. Quiz questions will also ask you to distinguish iterative and recursive processes, and to convert one to the other.

Some of the quiz questions may refer to the num-ways-to-form code from the “Recursion Exercises” programming assignment.

Readings

Concrete Abstractions, chapters 1 through 4. Exercises 1.3, 2.1, 2.3 through 2.15, 2.17, 2.23, 3.1, 3.2, 3.3, 3.6, 3.13 through 3.18, 3.20, 4.11, 4.12, 4.17, and 4.20.

Structure and Interpretation of Computer Programs, sections 1.1 through 1.2.2.

Suggested exercises

Concrete Abstractions: exercises 1.3, 1.7, 2.1, 2.3 through 2.15, 2.17, 2.23, 3.1, 3.2, 3.3, 3.5, 3.6, 3.13 through 3.18, 3.20, 4.11, 4.12, 4.17, and 4.20.

Structure and Interpretation of Computer Programs: all exercises in sections 1.1 and 1.2 except 1.13.

Sample questions for the “Functions and recursion” quiz

1. What does the function below return when given a positive integer as argument?

   (define mystery
    (lambda (n)
      (if (= n 0)
        0
        (+ 1 (mystery (quotient n 10)))) ) )

2. Explain why the function in exercise 1 generates a recursive process, and rewrite the function to generate an iterative process. Provide good comments for the arguments of any auxiliary functions you write.

3. Rewrite the function so that, given any integer argument, it returns a result analogous to what it returns with a positive integer argument.

4. How many calls to num-ways-to-form-with (see the “Recursion exercises” programming assignment) are made to evaluate the expression (num-ways-to-form 7)?

5. We can usually rewrite an if as a combination of and plus or by following this simple scheme: Replace (if test true-part false-part) with the equivalent expression (or (and test true-part) false-part). But this scheme fails for the expression (if (odd? 5) (even? 7) 'foo). Why does it fail? Suggest a more sophisticated way to rewrite if as a combination of ands and ors that does not fail.
Answers to sample questions for the “Functions and recursion” quiz

1. It returns the number of digits in the integer.

2. After each call except the first, the result must be added to the result of the remainder call. The following rewrite produces an iterative process.

   (define digit-count
   (lambda (n count-so-far)
     ; n represents the leftmost digits of the original argument to mystery
     ; count-so-far is the number of digits remaining in that argument
     (if (= n 0)
       count-so-far
       (digit-count (quotient n 10) (+ count-so-far 1)) )))

   (define mystery
   (lambda (n) (digit-sum n 0)) )

3. The mystery function results in an infinite recursion for negative arguments, and produces the wrong answer for 0, a one-digit number. Here’s a fix for both problems.

   (define mystery
   (lambda (n)
     (if (< n 0) (digit-sum (- n) 1) (digit-sum n 1)) )

   (define digit-count
   (lambda (n count-so-far)
     ; n represents the leftmost digits of the original argument to mystery
     ; count-so-far is the number of digits remaining in that argument
     (if (< n 10)
       count-so-far
       (digit-count (quotient n 10) (+ count-so-far 1)) )))

4. There are 29 calls, two with num-denominations equal to 5, two more with num-denominations equal to 4, two more with num-denominations equal to 3, three with num-denominations equal to 2, 3+8 = eleven with num-denominations equal to 1, and 2+7 = nine with num-denominations equal to 0.

5. If the test is true but the “true-part” is false, the and will fail and the “false-part” will be returned incorrectly in the rewritten version. A correct translation would be

   (or (and test true-part) (and (not test) false-part))
Program—Twenty-One

Goals
The main point of this program (along with the “Font Design” program later in the course) is the processing of functions as *data*. The information to be manipulated in this program consists of strategies for playing a simplified version of the “Twenty-One” game. A strategy is represented as a function, which is passed to another function to be applied. Strategies are built directly and constructed from other strategies.

Readings
*Concrete Abstractions*, chapter 5.
*Structure and Interpretation of Computer Programs*, section 1.3 through 1.3.4.

Related quizzes
Higher-order functions.

Assignment
Do the assignment on the following page.
The Twenty-One game

Background

For our purposes, the rules of Twenty-One are as follows. There are two players: the “customer” and the “dealer”. The object of the game is to be dealt a set of cards that comes as close to 21 as possible without going over 21. Cards have integer values between 1 and 10 inclusive, with 10 being four times as frequent as each other card value. The deck of cards contains an infinite number of each card.

Each player is dealt two cards, with one of the dealer’s cards face up. The dealer always takes another card—hits—if she has 16 or less, and always stops—stands—with 17 or more. The customer can play however she chooses, but must play before the dealer. If the customer exceeds 21—that’s called busting—she immediately loses (and the dealer doesn’t bother to take any cards). In case of a tie, neither player wins.
(These rules are substantially simplified from the version played in casinos. There is no “doubling down,” no “splitting,” no counting aces as 11, etc.)

Description of supplied code

For this assignment, the customer’s strategy of when to take another card is to be represented as a function. The function has two arguments: the customer’s total so far, and the dealer’s face-up card. The strategy function should return a Boolean value (#t or #f) that says whether or not the customer wants another card.

The file ~cs9d/lib/twenty-one.scm contains the code on the following page. Invoking (twenty-one~strategy) plays a game using the given strategy and a randomly shuffled deck, and returns 1, 0, or –1 according to whether the customer won, tied, or lost.
This plays a simplified game of Twenty-One.

(strategy customer-total-so-far dealer-up-card) should return #t or #f indicating whether the customer should hit. The twenty-one function then returns 1, 0, or -1 according to whether the customer won, tied, or lost.

(define (twenty-one strategy)

(result-of-dealer-play function assumes customer-total ≤ 21, and that the customer wants no more cards. It then plays the dealer's hand and returns the result of the game.

(define (result-of-dealer-play customer-total dealer-total-so-far)

(result-of-play function plays the customer's hand and then the dealer's hand if necessary, and returns the result of the game.

(define (result-of-play customer-total-so-far dealer-up-card)

(result-of-play (+ (random-card) (random-card)) (random-card) ) )

The random-card function returns a random card value between 1 and 10, inclusive. The value 10 is returned four times more often than any other value.

(define (random-card)

Code in ~/cs9d/lib/twenty-one.scm
Exercises

1. Define a strategy function stop-at-17 that’s identical to the dealer’s, i.e., it takes a card if and only if the customer’s total so far is less than 17. For example,

   (stop-at-17 16 any-dealer-card)

   should return #t, and

   (stop-at-17 17 any-dealer-card)

   should return #f.

2. Write a function result-of-n-plays such that

   (result-of-n-plays strategy n)

   plays n games with a given strategy and returns the number of games that the customer won minus the number that she lost. Use this to test your strategy from exercise 1, as well as strategies from the exercises that follow.

3. Define a strategy that hits (takes a card) if the dealer has a 1, 7, 8, 9, or 10 showing and the customer has less than 17, or the dealer has 2, 3, 4, 5, or 6 showing and the customer has less than 12. The strategy should stand (not take a card) otherwise. The idea is that in the second case, the dealer is much more likely to bust (go over 21), since there are more 10-point cards than anything else.)

4. Generalize exercise 1 above by defining a function stop-at. (stop-at n) should return a strategy that keeps hitting until a hand’s total is n or more. For example, (stop-at 17) is equivalent to the strategy in exercise 1:

   ((stop-at 17) 16 any-dealer-card)

   should return #t, and

   ((stop-at 17) 17 any-dealer-card)

   should return #f.

5. Define a function majority that takes three strategies as arguments and produces a strategy as a result, such that the result strategy always decides whether or not to hit by consulting the three argument strategies, and going with the majority. That is, the result should return #t if and only if at least two of the three argument strategies do. Using the three strategies from exercises 1, 3, and 4 as argument strategies, play a few games using the “majority strategy” formed from these three.
Miscellaneous requirements

To make sure your strategies do what you think they do, trace them when possible. Use the trace function to do this; given any number of function names, trace produces information about their argument values and values returned in subsequent calls. (The untrace function turns the output off.)

For grading, bring in a transcript showing (via trace) that your strategies have been called appropriately and that all possibilities for the strategies—i.e. all cases in any conditional expressions they contain—have been tested. On a UNIX account, the script command (described in an appendix) will produce such a transcript.

Don’t forget: a strategy is a function! Exercises 2 and 5 ask for functions that takes strategies as arguments; exercises 4 and 5 ask for functions that return strategies. If your stop-at function or your majority function returns a number or #t or #f, you’re doing the wrong thing.

As in the first programming assignment, you are to program in the functional style. Include comments with each of your functions that describe the function’s purpose and what arguments it expects.

Checklist

All exercises completed and tested appropriately.

A printed transcript of test results.

A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.

Reasonable names for functions and parameters.

No use of set!.
Quiz—Higher-order functions

Goals
A higher-order function is one that either takes a function as argument or returns a function as its value. Higher-order functions are powerful tools for generalization, as explained in the textbooks. Questions on this quiz involve writing, using, supplying pieces of, and analyzing higher-order functions.

Readings
Concrete Abstractions, chapter 5.
Structure and Interpretation of Computer Programs, section 1.3 through 1.3.4.

Suggested exercises
Concrete Abstractions: exercises 1.7 through 1.10, 5.1 through 5.23, and 6.2.
Structure and Interpretation of Computer Programs: exercises 1.29 through 1.46.
Sample questions for the “Higher-order functions” quiz

1. Find the values of the expressions

\[
\begin{align*}
((t \ s) \ 0) \\
((t \ (t \ s)) \ 0) \\
(((t \ t) \ s) \ 0)
\end{align*}
\]

where \( t \) and \( s \) are defined as follows:

\[
\begin{align*}
\text{(define s)} \\
\quad \text{(lambda (x) (+ 1 x)) )}
\end{align*}
\]
\[
\begin{align*}
\text{(define t)} \\
\quad \text{(lambda (f)} \\
\quad \quad \text{(lambda (x) (f (f (f x))))) ) )}
\end{align*}
\]

2. For each of the following expressions, provide a definition for \( f \) so that evaluation of the expression succeeds without causing an error.

\[
\begin{align*}
(f) \\
(f \ 3) \\
(((f)) \\
(((f)) \ 3)
\end{align*}
\]

3. Two functions appear below, one of which returns the next prime number greater than its argument, the other of which returns the next leap year following its argument.

\[
\begin{align*}
\text{(define (next-prime k)} \\
\quad \text{(if (prime? (+ k 1))} \\
\quad \quad (+ k 1) \\
\quad \quad \text{(next-prime (+ k 1)) ) )}
\end{align*}
\]
\[
\begin{align*}
\text{(define (next-leap-year year)} \\
\quad \text{(if (leap-year? (+ year 1))} \\
\quad \quad (+ year 1) \\
\quad \quad \text{(next-leap-year (+ year 1)) ) )}
\end{align*}
\]

Write a function that generalizes the two functions above, and show how to call it to produce the effect of next-prime and next-leap-year.

4. Write a function \text{maximizer} that, given as arguments two functions \( f \) and \( g \) that each takes an integer argument and returns an integer, returns a function that, when applied to an integer, returns the larger of the values that \( f \) and \( g \) would return.

5. What is the value of the following expression?

\[
\text{(let ((a +) \ (* 3)) (a \ *) )}
\]
Answers to sample questions for the “Higher-order functions” quiz

1. \(((t \ s) \ 0)\) returns 3, \(((t \ (t \ s)) \ 0)\) returns 9, and \(((t \ t) \ s) \ 0\) returns 27.

2. Answers will vary. For \((f)\) to evaluate without error, \(f\) must be a function with no arguments. For \((f \ 3)\) to evaluate without error, \(f\) must be a function with one argument. For \(((f))\) to evaluate without error, \(f\) must be a function of no arguments that returns a function of no arguments, for example

\[
\text{(define } f \\
\quad (\lambda ( ) (\lambda ( ) 5)) \text{)}
\]

For \(((\lambda) \ 3)\) to evaluate without error, \(f\) must be a function of no arguments that returns a function of no arguments that returns a function of one argument, for example

\[
\text{(define } f \\
\quad (\lambda ( ) (\lambda ( ) (\lambda (x) 5)))) \text{)}
\]

3. 
\[
\text{(define } \text{next-int } p \ k \\
\quad (\text{if } (p (+ k 1)) \\
\quad \quad (+ k 1) \\
\quad \quad (\text{next-int } p (+ k 1)) \text{)} \text{)}
\]

\[
\text{(define } \text{next-prime } k \\
\quad \text{(next-int prime? } k) \text{)}
\]

\[
\text{(define } \text{next-leap-year } \text{year} \\
\quad \text{(next-int leap-year? } \text{year}) \text{)}
\]

4. Here are two solutions:

\[
\text{(define } \text{maximizer } f \ g \\
\quad (\lambda (x) (\text{max } (f \ x) (g \ x))) \text{)}
\]

\[
\text{(define maximizer} \\
\quad (\lambda (f \ g) (\lambda (x) (\text{max } (f \ x) (g \ x)))) \text{)}
\]

5. Inside the \text{let}, \(a\)'s value is the + function and *'s value is 3. (* is just an identifier that can be bound to different values at different times.) Thus the value of the expression is 3+3 = 6.
Program—The Doctor program

Goals
For this assignment, you write a list-processing program. The context is a program that simulates a psychiatrist providing advice to a patient. The lists processed by the program are the patient’s input and the history of the interaction with the patient. You’ll get practice with techniques for recursively searching and restructuring lists.

Related quizzes
Evaluation and lists; list-processing recursion.

Readings
Concrete Abstractions, chapters 6, 7, and 8.
Structure and Interpretation of Computer Programs, sections 2.2 and 2.2.1 up through page 104; section 2.2.2 through page 111; sections 2.3 through 2.3.3.

Problem
Do the assignment on the following pages.
Background for the Doctor program

The Doctor program (originally called “Eliza”) was written in the early 1960’s by Joseph Weizenbaum of MIT. It simulates a Rogerian psychologist, in that it merely responds noncommittally to anything typed by the user.

The program is a relatively simple one. It doesn’t try to analyze the grammar of the user’s input. It does try, however, to put enough content into the response to make sense. Weizenbaum’s idea was to show that it didn’t take much intelligence or programming pizzazz for a program to act “intelligent”. Here are some of Doctor’s tricks.

a. It never gives only a fixed response in a given situation; it chooses randomly from a group of responses like “oh, that’s nice”, “please go on”, “please continue”, “many people have the same sorts of feelings”, etc. Some of these are concatenated to a version of the input: a reply to the comment “i am a loser” might be “why do you think you are a loser?” or “you feel that you are a loser”. It correctly translates “I” and “me” to “you” in these situations, but sometimes has trouble going from second person to first person (i.e. translating “you” to “I” or “me”).

b. Doctor also has a set of “trigger words” and responses to input containing those words. To a sentence mentioning one of the words “depressed”, “depression”, “depress”, or “suicidal”, it might respond “depression can be treated, you know” or “if you ever feel depressed, try drinking a glass of warm milk”. It knows about certain vulgar terms, and can respond with “please watch your language” or “same to you, fella”. Some of the responses will actually mention the trigger word, by having slots that get filled with the word that triggered that sentence; mentioning a family member, for instance, might result in the sentence “tell me more about your ___”, with the particular family member (mother, father, etc.) being substituted in the blank.

c. Doctor keeps a history of the patient’s remarks, and can respond if appropriate with comments like “earlier you said …”, “you do not seem very talkative today” (in response to several short answers), or “you seem to have a very negative attitude” (in response to several answers with the word “not” or “no”).

Problem

On the opposite side of this handout is the code for a (rather simple-minded) Doctor program. (It’s in the file doctor.scm in the ~cs9d/lib directory.) It assumes that input from the user and output to the user contains no punctuation, to avoid conflicts with Scheme operators. You are to improve the program by adding some of the features just mentioned:

- both features mentioned in a, that is, a response derived from the sentence typed by the user, and a response chosen randomly from a group of canned responses;
- response to mention of a family member as described in b, plus response chosen randomly for some other set of trigger words (you may choose which ones);
- one of the features mentioned in c, that is, some response that examines the history of sentences typed by the user.
You should check the patient’s history before you check for trigger words, and generate the responses mentioned in a above as a last resort. Don’t worry about the lack of punctuation.

The code we give you is not completely written in the functional paradigm, since reading and printing are essentially procedural activities. The code you add to it, however, should be completely functional; none of your code should compute a value that isn’t returned, or have any side effects.

For grading, prepare a sequence of user inputs that demonstrate all capabilities of your program. Your inputs should not only display all five types of response described above, but should also provide evidence that your program is generating responses with the proper priority (i.e. checking history before trigger words, and checking trigger words before generating a random response). Test your auxiliary functions individually and bring those tests for a tutor to inspect as well as your tests of the response function.

As usual, include comments with each of your functions that describe the function’s purpose and arguments.

**Useful Scheme functions**

The random, list-ref, and member functions will come in handy. They are described in both textbooks (*Structure and Interpretation of Computer Programs* describes a variant of member named memq). One may combine random and list-ref in a function named choose, which may then be used to return a response to vulgarity:

```scheme
(define (choose L)
  (list-ref L (random (length L))))

(choose
  '((same to you fella)
    (please watch your language)
    (you really feel strongly about this dont you) ) )
```

The member function is especially good to avoid long clumsy cond or or expressions. For example, the function

```scheme
(define (is-family-member? x)
  (cond
   ((equal? x 'father) #t)
   ((equal? x 'mother) #t)
   ((equal? x 'son) #t)
   ((equal? x 'daughter) #t)
   (else #f) ) )
```

may be coded much more concisely and clearly as

```scheme
(define (is-family-member? x)
  (member x '(father mother son daughter)) )
```

You will be expected in this assignment and in future assignments to use appropriate Scheme functions to avoid clumsy code.
Checklist

A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.

Thorough testing that produces all types of response, selected with proper priority.

A printed listing of function tests.

Individual testing of auxiliary functions.

Reasonable names for functions and parameters.

Avoidance of set! and of clumsy use of conditional expressions.
Framework for the Doctor program

; Start the program up by printing a welcome and
; returning the result of a visit with the doctor
; starting with an empty history (since nothing has happened yet).

(define (doctor)
  (show "please type all your input within parentheses.")
  (show " "); print a carriage return
  (show "Type (goodbye) to end the session.")
  (visit-with-doctor '())
)

; Return the result of a visit with the doctor.
; The history variable tells us some information about stuff
; the user has previously typed.
; All we do is print a prompt, read a list from the user
; (she has to parenthesize the input), compute a response, and
; return the result of subsequent interaction with the doctor.
; The response will have two parts: our actual reply to the
; user is the car, and the updated history is the cdr.
; This version of the program doesn’t collect a history.

(define (visit-with-doctor history)
  (show "")
  (display "** ")
  (let ((patient-says (read))) ; gets a list typed by the user
    (cond
      ((equal? (car patient-says) 'goodbye)
       '(your bill will be in the mail) )
      (else
       (let ((we-say (response patient-says history)))
        (show (car we-say))
        (visit-with-doctor (cdr we-say)) ) ) ) )
)

; Return a response to a sentence the patient has typed.
; The argument history represents a history of the dialog between us and the patient.
; Our response is a pair consisting of what we say to the patient (the car) and
; the updated history (the cdr).
; Note in this version we don’t change the history,
; and we merely parrot back what the user said.

(define (response sentence history)
  (cons sentence history))
Quiz—Evaluation with lists

Goals
This quiz tests your understanding of the details of the evaluation process, of the representation of lists, and of the built-in operations `list`, `cons`, and `append`. Questions ask you to evaluate and analyze Scheme expressions and to identify the box-and-pointer representation of a given list structure.

Readings
*Concrete Abstractions*, chapters 6 and 7. The `append` function doesn’t appear until page 218; you are better off just experimenting with it on the computer.

*Structure and Interpretation of Computer Programs*, sections 2.2 and 2.2.1 up through page 104; section 2.2.2 through page 111; sections 2.3 through 2.3.3.

Suggested exercises
*Concrete Abstractions*: exercises 7.1, 7.48, and 7.51.

*Structure and Interpretation of Computer Programs*: exercises 2.24, 2.25, 2.26, and 2.52 through 2.55.

The quote special form
The use of a quote mark in an expression, for example

```
'(a b c)
```

is actually a shorthand notation for the use of the quote special form

```
(quote (a b c))
```

Quote returns its argument unevaluated.
Sample questions for the “Evaluation with lists” quiz

1. Suppose that the append function is given the values (1 2) and ((3 4)) as arguments. What does it return?

2. Consider the function

   (define (quiz-f a b)
     (list a 'b '(list 'a b)))

   Evaluate the following expressions:
   (quiz-f 5 7)
   (quiz-f 'b 'a)
   (quiz-f '(a b) '(b a))

3. Insert parentheses and quotes in the following expression so that the result returned is the list ((january february)).

   cons january february

4. A CS 9D student, experimenting with Scheme evaluation, types the following to the interpreter, intending to apply cons to the arguments 1 and (2 3).

   (define f-list '(cons append list))
   ((car f-list) 1 '(2 3)) ; experiment

   Evaluating the “experiment” expression causes an error. Explain why, and indicate how the student should have defined f-list.

5. What is the box-and-pointer representation of the list ((january february))? 
Answers to sample questions for the “Evaluation with lists” quiz

1. (1 2 (3 4))

2. Evaluation of (quiz-f 5 7) returns (5 b (list 'a b)); evaluation of (quiz-f 'b 'a) returns (b b (list 'a b)); evaluation of (quiz-f '(a b) '(b a)) returns ((a b) b (list 'a b)).

3. The desired result is a list whose single element is (january february). This list is produced by providing that element and the empty list as inputs to cons as follows:

   (cons '((january february) '())

4. Elements of a quoted list are symbols, not functions. Thus the call to (car f-list) produces an error message something like “cons not a function”. The student should have defined the list as follows:

   (define f-list (list cons list append))

   Since all arguments to the list function are evaluated, this ensures that the list contains functions—the values bound to the names cons, list, and append—rather than symbols.

5. In *Concrete Abstractions*, the end of a list appears as

   january           february

   ( )
Quiz—List-processing recursion

Goals
This quiz tests your understanding of recursive functions that operate on lists. Questions ask you to invent, analyze, modify, and rewrite functions with arguments that are either linear lists or arbitrarily nested lists.

Readings
Concrete Abstractions, chapters 6, 7, and 8.
Structure and Interpretation of Computer Programs, sections 2.1 through page 104 except section 2.1.3; section 2.2.2 through page 111; sections 2.3 through 2.3.3.

Suggested exercises
Concrete Abstractions: exercises 6.8 through 6.18, 6.22 through 6.26, 7.2 through 7.4, 7.6 through 7.12, 7.15 through 7.21, 7.41, 7.45, 7.50, 7.51, 8.1 through 8.9, 8.12 through 8.20, 8.22 through 8.32.
Structure and Interpretation of Computer Programs: exercises 2.1, 2.2, 2.3, 2.7 through 2.19, 2.21, 2.22, 2.27 through 2.30, and 2.56 through 2.72.

Sample questions for the “List-processing recursion” quiz
1. Consider the following function.

   (define (mystery L k)
     (if (null? L)
         (list '( ) '( ))
         (let ((cdr-result (mystery (cdr L) k)))
           (if (< (car L) k)
               (list
                 (cons (car L) (car cdr-result))
                 (cadr cdr-result))
               (list
                 (car cdr-result)
                 (cons (car L) (cadr cdr-result)))))
         ))
     )

   What does evaluation of the expression (mystery '(1 9 4 5 2) 3) produce? What does mystery return in general?

2. Does the mystery function of exercise 3 generate a recursive process or an iterative process? If it generates a recursive process, rewrite it to generate an iterative process; if it generates an iterative process, rewrite it to generate a recursive process.
3. Write a function named `times-table` that, given two lists of integers, returns a list consisting of elements of the form \((n1 \times n2 = \text{product})\), where \(n1\) ranges over all values in the first list and \(n2\) ranges over all values in the second list. Thus the length of the result list should be the products of the lengths of the two argument lists. Here’s an example:

\[
\text{(times-table \(\'(2\ 3\) \(\'(4\ 5\ 6)\))}
\]

should return

\[
((2 \times 4 = 8) \ (2 \times 5 = 10) \ (2 \times 6 = 12) \\
(3 \times 4 = 12) \ (3 \times 5 = 15) \ (3 \times 6 = 18))
\]

4. The functions below are intended to “flatten” their generalized list argument, that is, essentially to return the result of all the parentheses except the outermost pair of parentheses. For example, \((\text{flat1 \(\'(a\ b) (((c)\ d))\ (e)\))}\) should return the list \((a\ b\ c\ d\ e)\). Fill in the blanks. (The `list?` predicate returns \#t if its argument is a list, \#f otherwise.)

\[
\text{(define (flat1 s)} \\
(\text{cond}) \\
\qquad (\text{((null? s) s)}) \\
\qquad (\text{((not (list? (car s))) ___ )}) \\
\qquad (\text{(else ___ )}) )
\]

\[
\text{(define (flat2 s)} \\
(\text{cond}) \\
\qquad (\text{((null? s) s)}) \\
\qquad (\text{((not (list? s)) ___ )}) \\
\qquad (\text{(else ___ )}) )
\]

5. How many recursive calls are made by each of the above routines when given the list \((a\ (b\ c)\ d)\) as an argument?
Answers to sample questions for the “List-processing recursion” quiz

1. Mystery is a partitioning function that splits the argument list into values less than the k argument and values greater than or equal to k. The returned result is

\[
((1 \ 2) \ (9 \ 4 \ 5))
\]

2. Mystery generates a recursive process since more computation occurs between the return of the recursive call and the return of mystery itself. Here’s an iterative implementation.

```scheme
(define (mystery L k)
  (helper L k '() '() )
)

(define (helper L k smalls larges)
  (cond
    ((null? L) (list smalls larges))
    ((< (car L) k) (mystery (cdr L) k (cons (car L) smalls) larges))
    (else (mystery (cdr L) k smalls (cons (car L) larges))) ) )
```

3. A nested recursion is appropriate for producing the desired result.

```scheme
(define (times-table L1 L2)
  (if (null? L1) '( )
    (append (helper (car L1) L2) (times-table (cdr L1) L2)) )
)

(define (helper value L)
  (if (null? L) '( )
    (cons
      (list value '* (car L) '=' (* value (car L)))
      (helper value (cdr L)) ) )
)
```

4. (define (flat1 s)
  (cond
    ((null? s) s)
    ((not (list? (car s))) (cons (car s) (flat1 (cdr s)))))
    (else (append (flat1 (car s)) (flat1 (cdr s))) ) )

(define (flat2 s)
  (cond
    ((null? s) s)
    ((not (list? s)) (list s))
    (else (append (flat2 (car s)) (flat2 (cdr s))) ) )
)

5. Flat1 makes 6 recursive calls (the initial call is not included). Flat2 makes 10 recursive calls, making an extra call for each symbol in the argument.
Program—Type inference

Goals
This assignment involves a function that operates on the text of another function, as
do compilers and interpreters.

Related quizzes
Evaluation and lists; list-processing recursion.

Readings
Concrete Abstractions, chapters 6, 7, and 8.
Structure and Interpretation of Computer Programs, sections 2.2 and 2.2.1 up through
page 104; section 2.2.2 through page 111; sections 2.3 through 2.3.3.

Problem
Do the assignment on the following pages.
Background for inferring types

_Concrete Abstractions_, on page 157, discusses the advantages of typed data. Scheme doesn't provide any builtin way to identify the type of a particular piece of information; an argument to a function might be a number, a list, or even another function, and data of the wrong type will be detected only when the function is run.

Languages such as Pascal and C++ make the programmer specify the type of each variable. Another approach to this problem, however, is _type inference_. If, for instance, a function includes the expression \((+ \ n \ k)\), one can infer that \(n\) and \(k\) have numeric values. Similarly, the expression \((f \ a \ b)\) indicates that the value of \(f\) is a function.

Problem

Write and test a function called _inferred-types_ that, given a definition of a Scheme function as argument, returns a list of information about the parameters of the function. The information list should contain one element per parameter; each element should be a two-element list whose first element is the parameter name and whose second element is a word indicating the type inferred for the parameter. Possible types are listed below.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>The type can't be inferred.</td>
</tr>
<tr>
<td>function</td>
<td>The parameter appeared as the first word in an unquoted expression or as the first argument of map or apply.</td>
</tr>
<tr>
<td>number</td>
<td>The parameter appeared as an argument of (+,-,), (*,/,=,&lt;,&gt;,&lt;=,&gt;=,) max, min, expt, sqrt, or round.</td>
</tr>
<tr>
<td>integer</td>
<td>The parameter appeared as an argument of quotient, remainder, even?, or odd?, or as the second argument of list-ref or list-tail.</td>
</tr>
<tr>
<td>list</td>
<td>The parameter appeared as an argument of append, car, cdr, cadr, caddr, cadddr, reverse, length; as the first argument of list-ref or list-tail; or as the second argument of map, cons, or member. (We're ignoring dotted pairs here.)</td>
</tr>
<tr>
<td>conflict</td>
<td>Conflicting types were inferred (note that an integer is also a number, so those types don't conflict).</td>
</tr>
</tbody>
</table>

You should assume for this problem that the body of the function to be examined does not contain any occurrences of \(if\) or \(cond\), although it may contain arbitrarily nested and quoted expressions. (A more ambitious inference function both would examine a more comprehensive set of functions and could infer conditions like “nonempty list”.)
Here’s an example of what your inference function should return.

```
(inferred-types
  '(define (foo a b c d e f)
    (f (append (a b) c '(b c)) (+ 5 d) (cons (car e) f)) ) )
```

should return

```
((a function) (b any) (c list) (d number)
 (e list) (f conflict))
```

**Miscellaneous requirements**

You don’t need to make your program work on both kinds of function definition. Just choose one—either `(define (f x) ...)` or `(define f (lambda (x) ...))` and tell the tutor which one you chose.

Test your program thoroughly, trying arguments that collectively provide all opportunities for conflict as well as all possible types. Your tests should include functions in which parameter names appear within quoted expressions to produce evidence that you’re handling quotes correctly. Test your auxiliary functions individually, and bring those tests for a tutor to see along with your tests of `inferred-types`.

As usual, provide comments with each of your functions that describe the function’s purpose and arguments.

**Checklist**

A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.

Thorough testing of all argument types that includes sufficient tests of type conflicts.

A printed listing of function tests.

Individual testing of auxiliary functions.

Reasonable names for functions and parameters.

Avoidance of `set!` and of clumsy use of conditional expressions.
Program—Font design

Goals
This assignment provides another variation on the “functions as data” theme. Here, characters in a digital font are represented as functions, and a line of characters to be displayed on the screen appear as a list of functions. Characters are combined via higher-order functions. Both textbooks have examples of collections of functions that operate on picture elements; the program you write for this assignment is similar. This program uses graphics primitives provided in the Gambit and stk interpreters.

Related quizzes
Evaluation and lists; list-processing recursion.

Readings
Concrete Abstractions, chapters 6, 7, and 8.
Structure and Interpretation of Computer Programs, sections 2.2 and 2.2.1 up through page 104; section 2.2.2 through page 111; sections 2.3 through 2.3.3. The appendices “Sequential programming with Scheme” and “Turtle graphics primitive procedures”.

Problem
Do the assignment on the following pages.
Background for font design

In the old days, printing used to be done with metal fonts, collections of character shapes that could be installed in racks on a printing press, covered with ink, and pressed onto paper. In modern printers, a font is a collection of procedures, one for each character in the font, written in a graphics description language. The printer contains a processor chip and a program that translates each procedure into the corresponding pattern of picture elements (also called “pixels”), which through various optical and chemical means is transferred to the paper. The old method of printing required differently-sized chunks of metal to print the same character in different sizes; increasing or decreasing the size of a character in a graphics description language, however, means only supplying a different value for the corresponding procedure parameter.

Computer fonts can also be used to display text on a workstation screen. Part 1 of this project involves writing and interpreting programs to draw characters on the screen. For part 2, you will write an interpreter for a simple text formatter.

Background for part 1

A character corresponds to a pattern of lines drawn in a grid whose y-coordinates range from 0 to 6. This line pattern will be represented here by turtle commands. Only a small subset of the turtle commands will be legal in this context:

    forward, penup, pendown, right, left.

Moreover, the argument to right or left must be 90, and the argument to forward must be greater than 0. (These restrictions allow you to focus on handling significant different operations, rather than having to write lots of essentially similar functions that handle operations that differ only slightly from one another. The restriction to right-angle turns saves you from having to do complicated trigonometric computations to keep track of the turtle position—this is a programming course, not a trigonometry course.)

Assuming that the turtle starts facing up, the following command sequence draws a plus sign:

    (penup) (forward 3) (right 90)
    (pendown) (forward 4) ; draw line from (0,3) to (4,3)
    (right 90) (right 90) (forward 2) ; move back to (2,3)
    (left 90) (forward 2) ; draw line from (2,5) to (2,1)
    (right 90) (right 90) (forward 4))

Turtle movement for this command sequence is as shown in the figure below.
The width of the plus sign is 4, the maximum distance moved by the robot in the x-direction relative to where it started.

Typically one wants characters in a variety of sizes—for example, a plus sign of size 4 for subscripted or superscripted expressions, a plus sign of size 8 for regular text expressions, and perhaps a plus sign of size 17 for formulas intended for overhead transparencies. Some examples are shown below.

```
+  +  +
```

It would be needlessly inefficient to have different sets of turtle commands for plus signs of various sizes; we want instead to represent a character as a pattern of turtle movements that are extended proportionally for characters of different sizes.

Thus for this project, a character will be represented as a procedure. The procedure is created by a make-character function that is given a sequence of turtle commands, the character pattern, as its argument. Drawing a character—invoking the corresponding function—involves taking its pattern and scaling and translating it to appear on the screen in a particular size and position. The character procedure takes two arguments:

- a scale factor that specifies the amount of magnification of the character pattern (2 means double, 10 means ten times, etc.), and
- the coordinates of an unscaled point at which to start drawing the character.

It will return the width of the character as displayed on the screen to provide a way to space characters naturally on a line.

Here is the definition of make-character:

```scheme
(define (make-character cmd-list)
  (lambda (scale point)
    (let ((current-x (xcor)) (current-y (ycor)))
      ; Move to point, then start drawing.
      (move-to (current-x (* scale (x-coord point)))
               (current-y (* scale (y-coord point))))
      (execute-each cmd-list scale)
      ; Draw the character, then go back where we started.
      (move-to current-x current-y)
      ; Return the furthest we got in the x direction.
      (find-max-x cmd-list scale (* scale (x-coord point)) UP) ) ) )
```
It is important to note that `make-character` doesn’t draw the character itself. It returns a procedure that, when invoked, draws a character. Thus one might say

```
(define plus-sign-list
  '((penup) (forward 3) (right 90)
   (pendown) (forward 4)
   (right 90) (right 90) (forward 2)
   (left 90) (forward 2)
   (right 90) (right 90) (forward 4)))
```

and then evaluate `((make-character plus-sign-list) 12 (make-point 0 0))` to display a plus sign 84 pixels high on the screen.

The `xcor` and `ycor` functions are among the turtle graphics primitives provided in the Gambit and stk environments. (They’re described in the appendix “Turtle graphics primitive procedures”.) The functions `x-coord`, `y-coord`, and `make-point` are the selectors and constructor for points, as described in exercise 6.26 in *Concrete Abstractions* and exercise 2.2 in Abelson and Sussman. `UP` represents the direction whose heading is 0 degrees. We provide you with the `move-to` procedure and a framework for code to execute each turtle command in a given list of commands. You write the `find-max-x` procedure.

A bunch of procedures may be found in the file `~cs9d/lib/font-design.scm`. Among them are a procedure to initialize the display:

```
(define (init-display)
  (clearscreen)
  (move-to LINE-START-X LINE-START-Y) )
```

The definitions

```
(define LINE-START-X -200)
(define LINE-START-Y 100)
```

are appropriate for Gambit. For stk, we think

```
(define LINE-START-X -200)
(define LINE-START-Y 150)
```

should work.

A line of characters is represented as a list of the corresponding procedures. It’s drawn on the screen using the `draw-line` procedure:

```
(define (draw-line-helper chars 0 DEFAULT-SIZE)
  (carriage-return DEFAULT-SIZE)
  (showturtle) )
```

```
(define character? procedure?)
(define DEFAULT-SIZE 12)
(define SPACE-BTWN-CHARS 2)
(define CHAR-HEIGHT 7)
```
; Draw the given characters on the given base line,
; relative to the current y position of the turtle,
; and scaled by the given scale factor.
(define (draw-line-helper chars base-line scale)
  (cond
   ((null? chars) #t)
   ((character? (car chars))
    (move-to ; this moves to the next char position on the line
     (+
       (xcor)
       ; procedure from the list is applied here to draw the character
       ((car chars) scale (make-point 0 base-line))
       (* SPACE-BTWN-CHARS scale) )
     (ycor) )
    (draw-line-helper (cdr chars) base-line scale) )
   (else "*** bad character in line") )
)

We move to the start of the next line by calling carriage-return:

(define (carriage-return scale)
  (move-to LINE-START-X (- (ycor) (* CHAR-HEIGHT scale)))
)

Recall that the procedure for a character included a call to a procedure execute-each
to execute the commands in make-character’s argument list. We supply you with part
of this code:

(define (execute-each cmd-list scale)
  (cond
   ((null? cmd-list) #t)
   (else
    (execute-cmd (car cmd-list) scale)
    (execute-each (cdr cmd-list) scale) ) )
)
(define (execute-cmd cmd scale)
  ((cadr
    (assoc
     (car cmd)
     (list ______________________ )) )))) )
Exercises for part 1

1. Design a square character that’s the same size as the plus sign defined in font-design.scm.

2. Fill in the blank in the execute-cmd procedure above without changing any of its code. The arguments you supply to the list procedure should produce a table associating command names with actual Scheme procedures. The table will be a list whose elements each have the form (symbol procedure). (To design the table, think first about how many arguments each of the procedures has.)

   The assoc function, given a symbol, returns the table element corresponding to the symbol. Assoc is built into Scheme, but if one were to define it, one would do so as follows:

   (define (assoc symbol table)
     (cond
      ((null? table) #f)
      ((equal? (caar table) symbol) (car table))
      (else (assoc symbol (cdr table))))
   )

   At this point you can supply a simple definition for find-max-x, say

   (define (find-max-x cmd-list scale x heading)
     (* 5 scale)
   )

   (this defines a “fixed-width” font) and use draw-line to display the plus sign and square character you’ve defined. Experiment with different widths to get a feel for the way in which the result returned by find-max-x is being used.

3. Write find-max-x. (This provides variable-width characters.) This will require choosing a representation for headings. Your choice may be either numeric or symbolic; be prepared to explain to a tutor why you chose one way rather than another.

4. Design a convincing set of test data for find-max-x. Be prepared to explain why your test data provides a convincing argument for find-max-x’s correctness.

5. Write and test a procedure called overlay that, given two characters, returns the result of superimposing one on the other. The width of the new character is the width of the wider of the two arguments. Overlaying the square you designed for exercise 1 with the plus sign defined in font-design.scm should yield a character that looks like

```
+ +
+ +
```
Overlay, like make-character, *returns* a character; it doesn’t draw the character.
Since the character is represented as a function, it won’t be drawn until it’s
applied to actual arguments.

6. Overlay is basically a one-line function. Explain what feature of characters allows
character overlaying to be implemented so easily.

7. Write and test a procedure called shift that, given a character and a point,
returns the character that results from shifting the argument character so that it
starts at the given point. The width of the result is the width of the argument
character plus the x coordinate of the point. The coordinate system for the argu-
ment to shift is the same as the coordinate system for the point argument to a
character. Shift, like overlay and make-character, returns a character; it doesn’t
draw the character itself.

8. Create an underbar character that’s one pixel high and five pixels wide (i.e. the
result of a (forward 4) command). Also create a vertical bar character that’s one
pixel wide and seven pixels high (the result of a (forward 6) command). Using
overlay and shift, create a T out of these two characters.

**Background for part 2**

The draw-line-helper procedure provided in the font-design.scm code is actually an in-
terpreter for a very simple letter description language, in which “programs” are sim-
ple lists of characters. For instance, you might have tested your solution to exercise 8
by saying

```
(draw-line-helper (list underbar vert-bar t) 0 DEFAULT-SIZE)
```

For the next two exercises, you will add code to draw-line-helper to extend the language
by interpreting more letter descriptions.

In the same way as the Scheme interpreter, given a Scheme expression, produces and
prints the value of the expression, your interpreter will be given a “line-language ex-
pression” as argument and will display the components of that expression. The first
argument to draw-line-helper will now be a list not merely of characters but of instruc-
tion information.

**Exercises for part 2**

9. Add the following instructions to the line-language interpreter.

   a. The word size. The next element of the argument list will be a positive integer.
The effect of this pair of elements is to change the scale factor to the given
integer for subsequent instructions.

   b. The word adjust-size. Again, the next element of the argument list will be an
integer. The effect of this pair of elements is to increase the scale factor by the
given integer for subsequent instructions (if the integer is negative, the scale
factor is decreased).
A sample argument list to the extended draw-line-helper might then be the following:

```
(draw-line-helper
  (list square 'size 24 square square 'adjust-size -10 square) 0 ; bottoms of characters sit on a base line at the current y pos
  DEFAULT-SIZE)
```

(You would ordinarily use draw-line instead, which calls draw-line-helper with 0 and DEFAULT-SIZE as the second and third arguments, and moves to the next line after drawing.) This call would draw a square of size 12, two squares of size 24, and a square of size 14. Make sure you know why the list procedure rather than a quote was used to construct the argument to draw-line-helper, and why size and adjust-size were quoted and square was not.

10. Now extend your interpreter to handle some additional instructions. Two of them involve adjusting the base line of subsequent characters, i.e. the y-coordinate on which the bottoms of the characters are aligned. Another is an explicit instruction to move to the start of the next line. Finally, there are two instructions that take instruction sequences as arguments. Here are the new instructions.

   c. The word superscript. The scale factor is divided by 2 and the base line is (additively) increased by \( \frac{\text{scale}^2}{3} \) for subsequent instructions. (I.e. the new base line is equal to the old base line + \( \frac{\text{scale}^2}{3} \).)

   d. The word subscript. The scale factor is divided by 2 and the base line is decreased by \( \frac{\text{scale}}{6} \) for subsequent instructions. (I.e. the new base line is equal to the old base line – \( \frac{\text{scale}}{6} \).)

   e. The word return. The effect is to move the turtle to the start of a new line, the height of which is determined by the current scale factor.

   f. A line-language expression, that is, a nested instruction sequence. Here’s an example:

```
(size 25 square (size 14 square square plus-sign) plus-sign)
```

Square and plus-sign are the procedures defined earlier. This example program could be itself nested inside some other program. After interpreting a program nested in this way, values for scale and base line should be restored to what they were prior to interpreting the nested program. Thus the above example would produce a display of a large square followed by two small squares and a small plus-sign, followed by a large plus-sign.

   g. The word repeat followed by a nonnegative integer followed by a line-language expression. The effect of this is to repeat the instructions in the given program the given number of times. A line-language expression that appears by itself has the same effect as preceding that expression by repeat 1.
Here’s another example that uses the square character of exercise 1.

\[
\text{(draw-line-helper}
\begin{align*}
\text{(list} & \quad \text{'repeat 2} \\
\text{\quad (list square (list 'subscript square) square 'return)} & \quad \text{square)} \\
\text{0} & \quad \text{DEFAULT-SIZE})
\end{align*}
\]

should draw three lines. The first two should each contain a square, a subscripted square, and another unsubscripted square; there should be less distance between the second and third squares on each line than between the first and second squares. The last line should contain a single unsubscripted square.

Tutors will ask you to test your code on a computer in the Self-Paced Center.

**Checklist**

- Solutions to all ten exercises:
  - a square character;
  - a completed execute-cmd procedure (with no changes to existing code);
  - a completed and tested find-max-x procedure;
  - thorough test data for find-max-x, with explanation of why the tests present convincing evidence for the correctness of the code;
  - a completed and tested overlay procedure;
  - an explanation of why overlay is so easy to implement;
  - a completed and tested shift procedure;
  - a T character created out of an underbar and a vertical bar character using shift and overlay;
  - implementation of the size and adjust-size instructions;
  - implementation of the superscript, subscript, return, and repeat instructions, along with nested instruction sequences.

A printed listing of your functions, indented to show their structure, accompanied by comments that describe their purpose and arguments.

A printed listing of function tests.

Reasonable names for functions and parameters.

Avoidance of set! and of clumsy use of conditional expressions.
Quiz—Higher-order functions with lists

Goals
This quiz, like the “Higher-order functions” quiz, involves the design, modification, and analysis of higher-order functions. Here, they include the built-in functions `map` and `apply`. Accumulation and filtering functions appear on this quiz, as well as other functions that process lists.

Readings
*Concrete Abstractions*, chapters 6, 7, and 8.
*Structure and Interpretation of Computer Programs*, pages 105, 106, and 112.

Suggested exercises
*Concrete Abstractions*: exercises 6.19, 7.5, 7.6, 7.9, 7.13, 7.14, 7.22, 7.23, 7.24, 7.40, 7.42, 7.46, 7.47, 7.49, 7.52, and 8.18.
*Structure and Interpretation of Computer Programs*: exercises 2.21 and 2.30 through 2.52.
**Sample questions for the “Higher-order functions with lists” quiz**

1. Suppose that you have a two-argument function \( f \) that you wish to test. In particular, you wish to evaluate it five times, supplying the list \((2 \ 3 \ 4)\) as the first argument to each evaluation, with the second argument ranging over values from the list \((1 \ 2 \ 3 \ 4 \ 5)\). Give a call to map that would test \( f \) as just described, collecting the results in a list.

2. Suppose now that you have a list of functions that you want to test, all on the same value. Write a function named test that’s given the list of functions and the value, and collects the results in a list as in exercise 1. The body of your test function should consist solely of a call to map.

3. Write a function accumulate that, given a list \( L \), a two-argument accumulating function \( f \), and a value init-value to return if \( L \) is empty, returns the result of accumulating \( f \) over the elements of \( L \) as follows:

\[
(f \ (\text{car } L) \ (f \ (\text{cadr } L) \ (f \ldots \ (f \ (\text{last-element-of } L) \ \text{init-value}) \ldots )))
\]

For example, the expression

\[
(\text{accumulate } \text{cons } '(1 \ 2 \ 3) \ '(\ ))
\]

should return the result of

\[
(\text{cons } 1 \ (\text{cons } 2 \ (\text{cons } 3 \ '(\ ))))
\]

namely \((1 \ 2 \ 3)\).

4. Write a function value-sum that, given a table each of whose elements has the form \((\text{symbol} \ \text{value})\), returns the sum of the values in the table. Neither value-sum nor any of the functions that it calls other than the accumulate function you wrote for exercise 2 should be recursive.

5. Write a function deep-reverse that, given a list \( L \) whose elements may themselves be lists, returns the result of reversing the elements of \( L \) as well as those in all sublists of \( L \). For example, \((\text{deep-reverse } '(1 \ (2 \ (3 \ 4) \ 5) \ (6 \ 7)))\) should return the list \(('(7 \ 6) \ (5 \ (4 \ 3) \ 2) \ 1)\). Your solution should include a call to map with deep-reverse as an argument.

6. A CS 9D student writes the function \( f \) below that, given three numbers and a list of functions, uses the first number to select a function from the list to apply to the other two numbers.

\[
(\text{define } (f \ \text{selector} \ n1 \ n2 \ f\text{-list})
\quad (\text{list-ref } f\text{-list} \ \text{selector} \ n1 \ n2))
\]

a. The student evaluates the following expression, intended to multiply 7 and 4.

\[
(f \ 2 \ 7 \ 4 \ '(+ \ - \ *))
\]

What happens? What call to \( f \) has the desired effect?

b. Rewrite \( f \) using apply.
Answers to sample questions for the “Higher-order functions with lists” quiz

1. 
   (map (lambda (x) (f '(2 3 4) x)) '(1 2 3 4 5))

2. 
   (define (test f-list x)
      (map (lambda (f) (f x)) f-list) )

3. 
   (define (accumulate f L init-value)
      (if (null? L) init-value
         (f (car L) (accumulate f (cdr L) init-value)) ) )

4. 
   (define (value-sum table)
      (accumulate + (map cadr table) 0) )

5. 
   (define (deep-reverse L)
      (if (not (list? L)) L
         (reverse (map deep-reverse L)) ) )

6. The evaluation produces an error message, since the elements of the argument list are symbols, not functions. A call that produces the desired behavior is

   (f 2 7 4 (list + - *))

Here’s f rewritten to use apply.

   (define (f selector n1 n2 f-list)
      (apply (list-ref f-list selector) (list n1 n2)) )
What’s functional programming all about?

Functional programming is programming with values. There are two kinds of values: builtin values (e.g. numbers), and values computed and returned by functions. When programming in a functional style, one writes functions that return a value and have no other effect. In almost all of CS 9D, you will be programming in this way.

In learning the discipline of functional programming, it is helpful to adjust one’s vocabulary, to think about the value a function returns rather than the action it performs. Here are some examples:

<table>
<thead>
<tr>
<th>procedural terminology</th>
<th>functional terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>differentiate</td>
<td>the derivative</td>
</tr>
<tr>
<td>translate</td>
<td>the translated form</td>
</tr>
<tr>
<td>make a move</td>
<td>the new game position</td>
</tr>
<tr>
<td>decide</td>
<td>the truth value of</td>
</tr>
<tr>
<td>parse</td>
<td>the structured version of</td>
</tr>
</tbody>
</table>

Programs in an imperative language like C are sequences of statements like x = x+1. Such statements are not used for their value, but for their side effects. Functional Scheme programs are collections of functions that are used in composition, as in mathematics.

What good is functional programming?

One advantage that results from functional programming is modularity. Functions without side effects don’t interfere with each other, and a change in the way a function computes its result should not affect any other part of the program.

Functional programming allows one to take advantage of any parallelism that’s provided in the underlying computer. For example, evaluation of the expression (f (g x) (h y z)) requires the evaluation of (g x) and (h y z). Since evaluation of g can have no effect on h and vice-versa, the evaluation may proceed in either order or even simultaneously.

Lastly, functional programming is a nice aid to mental organization, since it focuses one’s thinking on the single result to be returned by the function under consideration.
**Using script**

For all programming assignments in CS 9D, you will use a program called script to generate a transcript of your terminal session. Script’s argument is the name of a file that will contain everything typed between when you start script and you type exit. (script creates a new shell; exit terminates it.) A script command without an argument creates a file named typescript.

Confusion will probably result from typing another script command from within script. Confusion will also result from using a display editor like vi or emacs within script, since all the control characters that the editor outputs to manage the display will be output to the transcript.
Sequential programming in Scheme

Page 140 of Concrete Abstractions and page 86 of Structure and Interpretation of Computer Programs contain examples of sequential programming, that is, calling a sequence of procedures one after the next without regard for the values they return. This is typically done with procedures like display. Such procedures are only useful for their side effects, namely printing some characters on the screen; the values they return are typically meaningless.

Typically, a procedure you define has a single expression as its body. However, Scheme allows the definition of a procedure with more than one expression as the body, in the form

```
(define (proc ...) expression_1 expression_2 ... )
```

When such a procedure is called, the expressions are evaluated in sequence, but all their returned values except for the last are discarded. The value returned by the last expression is the value returned by the procedure. Here’s an example.

```
(define (proc x)
  (display x)
  (+ x 3)
  (+ x 5))
```

Evaluating `(proc 2)` results in 2 being printed on the screen, the expression `(+ 2 3)` being evaluated and the value discarded, and the expression `(+ 2 5)` being evaluated and the resulting value 7 being returned as the value of `proc`.

Like `display`, procedures that change the position or heading of the turtle are useful only for their side effects. They thus are appropriate for calling in this sequential programming style. For instance, the code we supply you for this project includes a procedure `move-to` that successively lifts the pen, moves to a given location on the screen, and points the turtle up:

```
(define (move-to x y)
  (penup)
  (setxy x y)
  (setheading 0) )
```

Scheme also allows multiple expressions to appear in a `let` expression and in a `cond` clause (though not in an `if`).
Turtle graphics primitive procedures

“Turtle graphics” procedures are available both in stk and in Gambit, although there are slight differences in the appearance of the graphics window between the two. Many of the graphics primitives have long and short names, as shown. The idea is that programs are more readable if you use the long names, but the short ones are convenient for interactive fooling around.

Things are drawn by a sort of graphics pointer called a “turtle” that has a specific position and orientation (angle) at any time. To draw a line from here to there, first you move the turtle to here, and then you put the turtle’s pen down and move it to there.1

The turtle is initially visible, with its pen down, in the center of the graphics window, facing upward. Its color depends on the version of Scheme:

stk: Background is black, initial pen color is white.
Gambit: Background is white, initial pen color is black.

(clearscreen)
(cs)

This procedure must be called before any other graphics procedures. Thereafter, it can be used to erase the graphics window and to reposition the turtle at the center of the screen facing up.

(forward dist)
(fd dist)

Move the turtle dist steps in the direction it’s facing. Draw a line if the pen is down; just move without drawing if the pen is up; erase if the eraser is down.

(back dist)
(bk dist)

Same as (forward (– dist)).

(right turn)
(rt turn)

Turn the turtle turn degrees clockwise. The turtle’s position doesn’t change, just its heading.

(left turn)
(lt turn)

Same as (right (– turn)).

(setxy newx newy)

Move the turtle to the specified absolute coordinates. The origin is in the center of the window; positive x is to the right; positive y is toward the top of the screen. Draws/erases/neither depending on the pen state, just like FORWARD.

1. The graphics interface embodied in these procedures was designed as part of the Logo programming language. Logo's syntax is different, but these are the official Logo procedure names, with the official meanings.
(setx newx)
Move the turtle horizontally to the specified absolute coordinate, leaving its vertical coordinate unchanged.

(sety newy)
Move the turtle vertically to the specified absolute coordinate, leaving its horizontal coordinate unchanged.

(setheading newh)
Turn the turtle to the specified absolute heading, measured in degrees clockwise from the positive y axis. That is, north is 0, east is 90, south is 180, and west is 270.

(home)
Same as (setxy 0 0) and (setheading 0).

(xcor)
Returns the turtle’s current horizontal position.

(ycor)
Returns the turtle’s current vertical position.

(pos)
Same as (list (xcor) (ycor)).

(heading)
Returns the turtle’s current heading.

(showturtle)
(st)
Maintain a visible display of the turtle’s position and heading in the form of a small triangle. The turtle’s position is in the center of the short side of the triangle, and its heading is toward the opposite vertex.

(hideturtle)
(ht)
Eliminate the visible display of the turtle. This speeds up line drawing at the cost of making it harder to figure out what’s going on.

(shown?)
Returns #t if the turtle is visible.

(pendown)
(pd)
From now on, when the turtle moves, it should draw a line.

(penup)
(pu)
From now on, when the turtle moves, it should leave no trace.
From now on, when the turtle moves, it should erase any dots it might happen to pass over.

Sets the color of the turtle (if shown) and its pen. The pen color only matters when the pen is down, but can be set regardless of the pen state. The color argument must be an integer $0 \leq \text{color} \leq 7$, with the following meanings:

<table>
<thead>
<tr>
<th>color</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>green</td>
</tr>
<tr>
<td>3</td>
<td>aqua</td>
</tr>
<tr>
<td>4</td>
<td>red</td>
</tr>
<tr>
<td>5</td>
<td>purple</td>
</tr>
<tr>
<td>6</td>
<td>yellow</td>
</tr>
<tr>
<td>7</td>
<td>white</td>
</tr>
</tbody>
</table>

Returns the current pen color.

On a UNIX account, you may print the contents of a graphics window by using the xwd and xpr commands:

```
xwd | xpr -device ps | lpr
```

The xwd command waits for you to click in a window, then sends a “dump” of the window to the xpr command, which converts it to Postscript and prints it on the default printer.
Emacs reference

by Drew Roselli ripping off Dan Garcia's idea and expanding upon it

Starting out

To begin, type `emacs` at the `%` prompt.

In the command lists below, C means the “control” key. To type the command C-x u, type the control key and x at the same time, then type u. M means the “meta” key. If your terminal doesn’t have a meta key, use the escape key followed by a brief pause.

<table>
<thead>
<tr>
<th>command</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-x u</td>
<td>Undo</td>
</tr>
<tr>
<td>C-g</td>
<td>Go away, or cancel a command</td>
</tr>
<tr>
<td>C-x C-c</td>
<td>Close session (save files, then exit emacs)</td>
</tr>
<tr>
<td>C-z</td>
<td>suspend emacs (or C-x C-z)</td>
</tr>
<tr>
<td>C-h</td>
<td>Help</td>
</tr>
<tr>
<td>?</td>
<td>“describe mode” (tell what the keys mean in the current mode)</td>
</tr>
</tbody>
</table>

Files

Emacs has auto-saving. Every 300 keystrokes, it will save the file you are editing into a temporary file called `#filename#`. In addition, every time you edit a file, a backup is made called `file~`.

When finding a file, use the tab key for filename completion and the space key for a list of possible completions.

<table>
<thead>
<tr>
<th>command</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-x C-f</td>
<td>Find (load) a file</td>
</tr>
<tr>
<td>C-x s</td>
<td>Save a file (or C-x C-s, but use C-x s)</td>
</tr>
<tr>
<td>C-x C-w</td>
<td>Write a file</td>
</tr>
<tr>
<td>C-x C-c</td>
<td>Close session (save files, then exit emacs)</td>
</tr>
</tbody>
</table>
Movement

Mouse and arrow keys will work if your terminal has them.

<table>
<thead>
<tr>
<th>command</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-f</td>
<td>Forward one character</td>
</tr>
<tr>
<td>C-b</td>
<td>Backward one character</td>
</tr>
<tr>
<td>C-n</td>
<td>Next line</td>
</tr>
<tr>
<td>C-p</td>
<td>Previous line</td>
</tr>
<tr>
<td>C-a</td>
<td>beginning of line</td>
</tr>
<tr>
<td>C-e</td>
<td>End of line</td>
</tr>
<tr>
<td>C-v</td>
<td>scroll up</td>
</tr>
<tr>
<td>C-d</td>
<td>Delete character</td>
</tr>
<tr>
<td>M-v</td>
<td>scroll down</td>
</tr>
<tr>
<td>M-&lt;</td>
<td>beginning of file</td>
</tr>
<tr>
<td>M-&gt;</td>
<td>end of file</td>
</tr>
<tr>
<td>M-&lt;</td>
<td>beginning of file</td>
</tr>
<tr>
<td>M-C-a</td>
<td>beginning of function</td>
</tr>
<tr>
<td>M-C-e</td>
<td>End of function</td>
</tr>
<tr>
<td>C-d</td>
<td>Delete character</td>
</tr>
<tr>
<td>M-d</td>
<td>Delete word</td>
</tr>
</tbody>
</table>

Cut and paste

Everything deleted or copied goes to the “Kill Ring” which you can retrieve (“yank back”) later. To define a region, set the mark at the beginning and position the cursor at the end.

<table>
<thead>
<tr>
<th>command</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-k</td>
<td>Kill line (copies to the kill ring)</td>
</tr>
<tr>
<td>C-SPC</td>
<td>set mark (or C-@)</td>
</tr>
<tr>
<td>C-w</td>
<td>Wipe region (kill region, copied to kill ring)</td>
</tr>
<tr>
<td>M-w</td>
<td>copy region (copy to kill ring)</td>
</tr>
<tr>
<td>C-y</td>
<td>Yank back last kill</td>
</tr>
<tr>
<td>M-y</td>
<td>Yank pop (yank back previous kill from kill ring)</td>
</tr>
</tbody>
</table>
## Windows and buffers

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-x 2</td>
<td>split window vertically</td>
</tr>
<tr>
<td>C-x 5</td>
<td>split window horizontally</td>
</tr>
<tr>
<td>C-x 1</td>
<td>close all other windows</td>
</tr>
<tr>
<td>C-x 0</td>
<td>kill the window the cursor is in</td>
</tr>
<tr>
<td>C-x o</td>
<td>other window (put cursor in the other window)</td>
</tr>
<tr>
<td>C-x b</td>
<td>switch Buffers</td>
</tr>
<tr>
<td>C-x k</td>
<td>Kill buffer</td>
</tr>
</tbody>
</table>

## Miscellany

I-searches are incremental searches—that is, emacs searches for the pattern as you type. When you find what you are looking for, type the escape key.

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C-s</td>
<td>i-search forward</td>
</tr>
<tr>
<td>C-r</td>
<td>i-search backward</td>
</tr>
<tr>
<td>M-%</td>
<td>query replace</td>
</tr>
<tr>
<td>M-;</td>
<td>go to line</td>
</tr>
<tr>
<td>C-x m</td>
<td>mail</td>
</tr>
<tr>
<td>M-x rmail</td>
<td>read mail (also send and reply)</td>
</tr>
<tr>
<td>M-x shell</td>
<td>will start a Unix shell in emacs</td>
</tr>
<tr>
<td>M-x man</td>
<td>manual entry (same as man, but in editor mode, not more mode)</td>
</tr>
<tr>
<td>M-x run-lisp</td>
<td>run a Lisp listener in a buffer</td>
</tr>
<tr>
<td>M-x command-name</td>
<td>general way to execute a command</td>
</tr>
</tbody>
</table>

## Useful commands for interacting with C

<table>
<thead>
<tr>
<th>Command</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>M C-\</td>
<td>indent region (according to C convention)</td>
</tr>
<tr>
<td>M-x compile</td>
<td>compiles</td>
</tr>
<tr>
<td>C-x `</td>
<td>(i.e. control-x backquote) next error (will find your compiler errors for you)</td>
</tr>
<tr>
<td>M-x gdb</td>
<td>runs the gdb debugger and follows the source code with an =&gt; in another window</td>
</tr>
</tbody>
</table>
How to learn more

The online help system is extensive. Typing C-h (help) will offer you several different kinds of help. Type C-h again for help with help, etc. Some of the more useful options are

- **b**  describe all the key bindings
- **i**  info mode. This is the entire emacs manual in hypertext (it contains its own instructions).

Customization

Emacs allows you to create a .emacs file and enter customizations such as the following. The .emacs file is executed when emacs is invoked. Customizations can be set differently for different modes. Modes are set by the file name, i.e., any file ending with .c will be assumed to be a C program file and will invoke the C mode automatically. A sample .emacs file appears on the next page.
Sample .emacs file

(setq inhibit-startup-message t) ; don't print emacs startup message
(setq default-major-mode text-mode)

; rebind some keys for convenience
(global-set-key "\M-h" 'help-for-help) ; reset help to M-h key.
(global-set-key "\C-h" 'delete-backward-file) ; C-h (backspace) deletes
(global-set-key "\M-r" 'rmail) ; read mail mode
(global-set-key "\M-m" 'mail)
(global-set-key "\M-s" 'save-buffer)
(global-set-key "\M-g" 'goto-line)
(global-set-key "\M-c" 'compile)
(global-set-key "\M-q" 'query-replace)
(global-set-key "\M-i" 'indent-region) ; to avoid typing “make” or “compile”

; set default indentations for C mode
(defconst c-indent-level 4)
(defconst c-brace-offset 0)
(defconst c-arg-decl-indent 0)
(defconst c-continued-statement-offset 4)
(defconst c-auto-newline nil)
(defconst c-label-offset -2)

; set return to indent automatically while in C mode
(setq c-mode hook (function (lambda nil
  "Customize c-mode"
  (interactive)
  (setq-indent-tabs-mode t)
  (define-key "\r" 'c-newline-indent)
  (define-key "\n" 'c-newline-indent)
  (define-key c-mode-map "\n" 'c-newline-indent)))

(defun c-newline-indent (arg)
  "user defined function to make new line and indent"
  (interactive "P")
  (newline arg)
  (c-indent-line))

(setq-default compile-command "make")
Notes from tutoring sessions
Notes from tutoring sessions
Notes from tutoring sessions
Notes from tutoring sessions
Notes from tutoring sessions