Introduction

In CS 47A, you learn about a variety of ways to organize programs and data, with emphasis on the mechanisms needed to support these organizations. Our intent is to expand your programming toolkit, addressing the saying that "when all you have is a hammer, everything looks like a nail"; we hope you emerge from CS 47A with a wrench, pliers, and screwdriver in addition to whatever hammers you've learned about already.

CS 9D and 61B are enforced prerequisites; reference will occasionally be made to topics covered and assignments included in CS 9D in particular. CS 9D and 47A together satisfy all EECS department requirements for CS 61A.

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).
- *Structure and Interpretation of Computer Programs*, Harold Abelson *et al*. (MIT Press, 1996).

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.

Structure of quizzes and programs in CS 47A

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.

group	*programming assignments*	*quizzes*
a	Orientation Rule-based programming	Pattern matcher
b	Generic operations Iterators and lazy evaluation exercises	Data representation mechanisms Streams
c	Scheme interpreter exercises	Scheme interpreter
d	Implementation of object-oriented programming	Classes in Scheme

Note that this breakdown is different from what's required to satisfy course deadlines. For information about deadlines, consult the "Information and Regulations" document.

Program—Orientation

Goals

In this assignment, you run the program used 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. Complete it as early in the semester as possible.

Program—Rule-based programming

Goals

In this assignment, you reimplement your type inference program using *rules*, that is, pattern-action pairs.

Related quizzes

Higher-order functions with lists; pattern matcher.

Readings

Concrete Abstractions, section 7.6.

The appendix “Rule-based programming”.

Problem

Do the assignment on the following pages.

Revised type inference

On the following pages is code from *Concrete Abstractions* that implements a rule-based query system, plus a set of functions that allow you to recode your type inference program as a rule-based application. (The code appears in the files matches.scm and rule.based.type.fw.scm, both in the ~cs9d/lib directory.) Complete the program.

The main component to be added to the framework code is the table of rules, named type-inference-p/a-list. This table will contain a rule for each possible type of expression. (Our solution has twelve rules.) Each rule combines a pattern with which a given expression type is recognized with a function that's called on an expression that matches the pattern. Unlike the query rules discussed in *Concrete Abstractions*, the action parts of these rules will be functions that call expr-inferred-types recursively on subexpressions. Note from the expr-inferred-types function that the action part of each rule is applied to two arguments: the result of a pattern match and the accumulated list of symbols and their associated type information.

Incorporating calls to display and newline in the action parts of your rules is a good debugging technique.

Requirements for this assignment are otherwise identical to those of the earlier type inference assignments. As in that assignment, you should 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 auxiliary functions individually.

The framework code assumes that functions are defined in the form

```
(define (f x) ...)
```

You may change the param-list and body functions if you wish to process functions defined using lambda. Otherwise, make no other changes to the framework functions.

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

No changes to the framework functions other than those specified.

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

A 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.

Concrete Abstractions query code

; Constructors/accessors for a rule.

```
(define make-pattern/action
   (lambda (pattern action)
      (cons pattern action) ) )

(define pattern car)
(define action cdr)
```

; Return true exactly when the pattern matches the question.
; The pattern and the question are both lists.
; Each element of the pattern is either
;   the symbol _, which matches any single element;
;   the symbol ..., which must occur at the end of the pattern
;   and matches the remainder of the question;
;   a list, which matches any element of the list;
;   some other symbol, which matches itself.
; This is the version from Concrete Abstractions page 198, augmented as in exercise 7.29.

```
(define (matches? pattern question)
   (cond
      ((null? pattern) (null? question) )
      ((null? question) #f)
      ((list? (car pattern))
       (if (member (car question) (car pattern))
         (matches? (cdr pattern) (cdr question))
         #f) )
      ((equal? (car pattern) '...) #t)
      ((equal? (car pattern) '_)
       (matches? (cdr pattern) (cdr question)) )
      ((equal? (car pattern) (car question))
       (matches? (cdr pattern) (cdr question)) )
      (else #f) ) )
```

; Return the list of elements in the question that match ... and _ in the pattern.
; What matches ... is always a list.
; Example: `(substitutions-in-to-match '(a _ b _ ...) '(a x b (1 2) g h))`
; returns `(x (1 2) (g h))`.
; Precondition: `(matches? pattern question)`.

```
(define (substitutions-in-to-match pattern question)
  (cond
    ((null? pattern) '( ))
    ((null? question) '(( )))  ; (car pattern) must be ...
    ((list? (car pattern))
     ; We know (member (car question) (car pattern)).
     (substitutions-in-to-match (cdr pattern) (cdr question)) )
    ((equal? (car pattern) '...) (list question))
    ((equal? (car pattern) '_)
     (cons
       (car question)
       (substitutions-in-to-match (cdr pattern) (cdr question)) ) )
    ; We know (equal? (car pattern) (car question)).
    (else
     (substitutions-in-to-match (cdr pattern) (cdr question)) ) ) )
```

Rule-based type inference framework

```
; The list of pattern-action pairs.
(define type-inference-p/a-list '( ))

; Useful accessors for dealing with chunks of Scheme code.

(define first car)
(define second cadr)
(define third caddr)

(define (param-list f) (cdr (second f)) )
(define (body f) (third f))

; Return types inferred for the parameters of the given function.

(define (inferred-types f)
  (expr-inferred-types
    (body f)
    'any
    (map (lambda (symbol) (list symbol 'any)) (param-list f)) ) )

; Return types inferred for the symbols in the symbol list
; in the given expression (which is itself of type expr-type).
; Do this by querying the rule data base to find the rule that
; matches the expression, and then applying that rule to process
; the subexpressions.

(define (expr-inferred-types expr expr-type symbol-list)

   (define (loop p/a-list)
      (cond
         ((null? p/a-list) #f)
         ((matches? (pattern (car p/a-list)) expr)
          ((action (car p/a-list))
            (substitutions-in-to-match (pattern (car p/a-list)) expr)
            symbol-list) )
         (else (loop (cdr p/a-list))) ) )

   (if (not (list? expr))
      (if (assoc expr symbol-list)
         (augmented symbol-list expr expr-type)
         symbol-list)
      (loop type-inference-p/a-list) ) )

; Return the result of including the given type requirement for the given symbol
; in the symbol list.

(define (augmented symbol-list symbol type)
   (if (eq? symbol (first (car symbol-list)))
      ; We found the symbol in the table.
      ; Incorporate its current type requirement into the type information already seen.
      (cons
         (list symbol (type-sum type (second (car symbol-list))))
         (cdr symbol-list))
      (cons
         (car symbol-list)
         (augmented (cdr symbol-list) symbol type) ) ) )
```

```
; Return the result of resolving the two argument types.
;    x+x = x
;    any+x = x+any = x
;    number+integer = integer+number = number
;    x+y = CONFLICT

(define (type-sum new-type old-type)
   (cond
      ((eq? new-type old-type) new-type)
      ((eq? new-type 'any) old-type)
      ((eq? old-type 'any) new-type)
      ((and (eq? new-type 'number) (eq? old-type 'integer)) 'integer)
      ((and (eq? new-type 'integer) (eq? old-type 'number)) 'integer)
      (else 'CONFLICT) ) )
```

Quiz—Pattern matcher

Goals

This quiz tests your familiarity with the pattern matcher component of the movie query system described in *Concrete Abstractions*, section 7.6. The version tested is the one listed in the "Rule-based programming" assignment, which handles "_" and list wild cards as described on page 198 of *Concrete Abstractions*. You will be asked to identify patterns that represent given queries, to analyze segments of the pattern matching code, and to add features to the `matches` and `substitutions-in-to-match` functions.

Readings

Concrete Abstractions, section 7.6.

Suggested exercises

Concrete Abstractions: exercises 7.25, 7.26, 7.30, 7.34, 7.35, 7.37, 7.38.

Sample questions for the "Pattern matcher" quiz

1. Describe all patterns `p` and questions `q` for which the length of `p` is not equal to the length of `q`, yet `(matches p q)` returns `#t`.
2. Modify the `substitutions-in-to-match` function so that, instead of returning a list of lists, it returns a flat list whose elements include *terminator symbols*—occurrences of the symbol "`|`"—that indicate the end of each sequence of symbols matched by a wild card. For example, instead of returning

   ```
   ((movie) (was) (1982))
   ```

   the modified version should return

   ```
   (movie | was | 1982 |)
   ```

3. Give a call to the version of `substitutions-in-to-match` of exercise 2 that returns the list

   ```
   (director | the godfather |)
   ```

Answers to sample questions for the “Pattern matcher” quiz

1. The specified set of pattern/question pairs can be described as follows:

   The pattern contains exactly one occurrence of the symbol “...”, as its last element.

   If n is the length of the pattern, then the first n–1 elements of the question are the same as the first n–1 elements of the pattern, and the question contains at least two additional elements.

2. The substitutions-in-to-match may be rewritten as follows:

```
(define (substitutions-in-to-match pattern question)
   (cond
      ((null? pattern) '( ))
      ((null? question) '( | ))
      ((list? (car pattern))
       (substitutions-in-to-match (cdr pattern) (cdr question)) )
      ((equal? (car pattern) '...) (append question '( | )))
      ((equal? (car pattern) '_)
       (cons
         (car question)
         (cons
            '|
            (substitutions-in-to-match (cdr pattern) (cdr question)) )))
      (else
         (substitutions-in-to-match (cdr pattern) (cdr question)) ) ) )
```

3. Here’s a call that returns the specified result:

```
(substitutions-in-to-match
   '(who is the _ of ...)
   '(who is the director of the godfather))
```

Program—Generic operations

Goals

This assignment addresses ways to implement *generic operations* common to multiple data types.

Related quizzes

Data representation mechanisms.

Readings

Concrete Abstractions, chapter 9.

Structure and Interpretation of Computer Programs, chapter 2, especially sections 2.2.4, 2.4, and 2.5.

Problem

This assignment is described on the following pages.

Generic operations

Background

In your data structures course (a prerequisite for CS 47A), you learned that a particular data type may be implemented in a variety of ways, depending on what resources—memory or execution time—one wishes to optimize. Here we deal with the problem of using more than one implementation of a data type simultaneously, thereby using operations that are common, or *generic*, to the implementations. In particular, we consider two ways to implement a *set of integers* data type that supports four operations:

> constructing a set from a list of integer elements (new-set);
> checking if a given integer is an element of the set (member?);
> returning the result of adding another integer to the set (with-element);
> returning a list of the set elements (elements).

One of the representations will be as a simple list of integers. The other will (potentially) economize on memory, storing the integers as a list of *intervals*; each interval contains the first and last element of a sequence of consecutive integers. Some way is needed to distinguish instances of the two representations. We thus start by attaching a *type* symbol to each representation: either list or intvls.

We then move to organizing the operations. One way is to have "smart operations" that determine from the data type what to do, for example,

```
(define (member? x set)
   (cond
      ((list-set? set) (list-member? x set))
      ((intvls-set? set) (intvls-member? x set)) ) )
```

Adding a third data type requires adding another case to all the generic functions.

Another solution is to put all the decisions into a table; each generic operation would then access the table in a uniform way to figure out what to do. Here's an example:

```
(define (member? x set)
   ((operate 'member? set) x) )
```

Abelson and Sussman describe an operate function that uses a keyword for the operation, plus the type from the set, to retrieve the appropriate membership function from the table. In *Concrete Abstractions*, each type contains its own table. Assuming that a change to the program will involve introducing another implementation of the set type, this organization is preferable. It requires modification only of the initialization function for the new type to set up new table entries.

A third approach is "smart data" that autonomously respond to *messages*. One kind of set would respond to the "member?" message in one way, and another kind of set would respond in another way. In this message-passing style, each data type is represented as a function[1] that takes a single symbol—the message—as argument and returns a function that handles that message.

1. Abelson and Sussman call it a *dispatch* procedure.

Problem

This is a three-part assignment.

Part 1

The file ~cs47a/lib/sets.fw.scm contains code that implements a set of integers as a typed list of integers, plus function headers for an implementation of a set of integers as a typed sorted list of intervals. Supply the bodies for the second implementation. At least one of your operations must work entirely with the internal list-of-intervals representation; that is, neither it nor any functions that it calls may convert the list of intervals to a list of integers.

Do not change any of the existing code in sets.fw.scm. Use the expressions in the file ~cs47a/lib/set.tests.scm to test your code.

Part 2

The file ~cs47a/lib/dd.sets.fw.scm contains a modified version of the functions from page 260 of *Concrete Abstractions* that implement data-directed generic operations. First, study the operate function and the generic functions so that you know what the contents of each table (one for the list-of-integers implementation and the other for the list-of-intervals implementation) will be. Then modify the functions from part 1 appropriately. Finally, supply the arguments for the calls to make-type for list-set-type and intvls-set-type to complete the data-directed implementations.

Do not change any of the existing code in dd.sets.fw.scm. Use the expressions in the file ~cs47a/lib/dd.set.tests.scm (almost identical to set.tests.scm) to test your code.

Part 3

The file ~cs47a/lib/mp.sets.fw.scm contains a message passing implementation of a set of integers as a simple list of integers. Again, study the generic operations so that you know what the dispatch function will return in each case. Then supply the body of the function new-intvls-set that uses message passing to implement a set of integers as a sorted list of intervals.

Do not change any of the existing code in mp.sets.fw.scm. Use the expressions in the file ~cs47a/lib/dd.set.tests.scm to test your code.

Miscellaneous requirements

You may have learned that Scheme contains a set! function that acts like an assignment statement in C, Pascal, or whatever. Don't use it. Your code should be free of side effects.

Checklist

All parts implemented as specified:

completion of the list-of-intervals implementation without changing any of the existing code, implementing at least one of the elements, with-element, and member? functions without using any other structure for the represented integers;

completion of the data-directed implementations without changing any of the existing code;

completion of the message-passing version of the list-of-intervals implementation without changing any of the existing code.

A transcript of test results.

A 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!.

sets.fw.scm

```
; Functions that implement the set-of-integers type
; using an unordered list as the underlying representation.
; Each set thus represented has the symbol "list" as its first element
; and the set members as the remaining elements.

; Type name.
(define list-set-type 'list)

; Return the result of forming a set out of the given elements.
(define (new-list-set elements)
  (tagged-datum list-set-type elements) )

; Return true if x is in the set, false otherwise.
(define (list-member? x set)
  (member x (contents set)) )

; Return the result of adding x to the set.
(define (list-with-element x set)
  (if (list-member? x set)
      set
      (tagged-datum list-set-type (cons x (contents set))) ) )

; Return a list of the elements of the set.
(define (list-elements set)
  (contents set) )

; Functions that implement the set-of-integers type using a
; sorted list of intervals as the underlying representation.
; Each set thus represented has the symbol "intvls" as its first element,
; with each remaining element being a two-element list representing
; all the integers between the car of the list and the cadr, inclusive.
; The intervals of integers are sorted, e.g. the list
;   ((5 8) (10 10) (14 14) (100 104))
; represents the set {5, 6, 7, 8, 10, 14, 100, 101, 102, 103, 104}.
; Moreover, consecutive intervals don't overlap.

; Type name.
(define intvls-set-type 'intvls)

; Return the result of forming a set out of the given elements.
(define (new-intvls-set elements)
  ... )

; Return true if x is in the set, false otherwise.
(define (intvls-member? x set)
  ... )

; Return the result of adding x to the set.
(define (intvls-with-element x set)
  ... )

; Return a list of the elements of the set.
(define (intvls-elements set)
  ... )
```

```
; Functions to implement tagged data (from Concrete Abstractions)

(define (tagged-datum type value)
  (cons type value) )

(define type car)

(define contents cdr)

; Generic operations.

(define (new-set elements type)
  (cond
    ((equal? type list-set-type) (new-list-set elements))
    ((equal? type intvls-set-type) (new-intvls-set elements))
    (else (error "unknown initialization type: " type)) ) )

(define (member? x set)
  (cond
    ((list-set? set) (list-member? x set))
    ((intvls-set? set) (intvls-member? x set))
    (else (error "unknown type in member?: " (type set))) ) )

(define (with-element x set)
  (cond
    ((list-set? set) (list-with-element x set))
    ((intvls-set? set) (intvls-with-element x set))
    (else (error "unknown type in with-element: " (type set))) ) )

(define (elements set)
  (cond
    ((list-set? set) (list-elements x set))
    ((intvls-set? set) (intvls-elements x set))
    (else (error "unknown type in elements: " (type set))) ) )

(define (list-set? set)
  (equal? (type set) list-set-type) )

(define (intvls-set? set)
  (equal? (type set) intvls-set-type) )
```

set.tests.scm

```
; Some sample manipulations of sets.
; Note that any true (non-#f) value is acceptable anywhere #t appears.

(define s1 (new-set '(7 2 13 11 6 3) list-set-type))
(member? 11 s1)         ; should be #t
(member? 12 s1)         ; should be #f
(elements s1)

(define s2 (with-element 11 s1))    ; should be the same set as s1
(member? 11 s2)         ; should be #t
(member? 12 s2)         ; should be #f
(elements s2)

(define s3 (with-element 12 s1))    ; one more element than s1 has
(map (lambda (x) (member? x s3)) '(11 12 14))   ; should be (#t #t #f)
(elements s3)

(define s4 (new-set '(7 2 13 11 6 3) intvls-set-type))
; the following should return (#f #t #t #f #f #t #t #f #f #f #t #f #t #f)
(map
  (lambda (x) (member? x s4))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s4)

(define s5 (with-element 2 s4))    ; should be the same set as s4
; the following should return (#f #t #t #f #f #t #t #f #f #f #t #f #t #f)
(map
  (lambda (x) (member? x s5))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s5)

(define s6 (with-element 1 s4))
; the following should return (#t #t #t #f #f #t #t #f #f #f #t #f #t #f)
(map
  (lambda (x) (member? x s6))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s6)

(define s7 (with-element 4 s4))
; the following should return (#f #t #t #t #f #t #t #f #f #f #t #f #t #f)
(map
  (lambda (x) (member? x s7))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s7)

(define s8 (with-element 5 s4))
; the following should return (#f #t #t #f #t #t #t #f #f #f #t #f #t #f)
(map
  (lambda (x) (member? x s8))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s8)
```

```
(define s9 (with-element 9 s4))
; the following should return (#f #t #t #f #f #t #t #f #t #f #t #f #t #f)
(map
  (lambda (x) (member? x s9))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s9)

(define s10 (with-element 12 s4))
; the following should return (#f #t #t #f #f #t #t #f #f #f #t #t #t #f)
(map
  (lambda (x) (member? x s10))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s10)

(define s11 (with-element 14 s4))
; the following should return (#f #t #t #f #f #t #t #f #f #f #t #f #t #t)
(map
  (lambda (x) (member? x s11))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s11)

(define s12 (with-element 15 s4))
; the following should return (#f #t #t #f #f #t #t #f #f #f #t #f #t #f #t #f)
(map
  (lambda (x) (member? x s12))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16))
(elements s12)

(define s13 (with-element 8 s9))
; the following should return (#f #t #t #f #f #t #t #t #t #f #t #f #t #f)
(map
  (lambda (x) (member? x s13))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s13)

(define s14 (with-element 10 s9))
; the following should return (#f #t #t #f #f #t #t #f #t #t #t #f #t #f)
(map
  (lambda (x) (member? x s14))
  '(1 2 3 4 5 6 7 8 9 10 11 12 13 14))
(elements s14)
```

dd.sets.fw.scm

```
; Functions that implement data-directed programming (from Concrete Abstractions).

(define (make-type name operation-table)
    (cons name operation-table) )

(define type-name car)

(define type-operation-table cdr)

(define (operate operation-name value)
  (table-find
    (type-operation-table (type value))
    operation-name
    (lambda (procedure) (procedure (contents value)))
    (lambda ( )
      (error
        "No way of doing operation on type"
        operation-name
        (type-name (type value)) ) ) ) ) )

(define (make-table keys values)
  (map list keys values) )

(define (table-find table key what-if-found what-if-not)
  (let ((result (assoc key table)))
    (if result (what-if-found (cadr result)) (what-if-not)) ) )

; *****
; Set up of types for data-directed programming.

(define list-set-type
  (make-type
    'list
    ... ) )

(define intvls-set-type
  (make-type
    'intvls
    ... ) )

; *****
; Generic operations on sets.

(define (member? x set)
  ((operate 'member? set) x) )

(define (with-element x set)
  ((operate 'with-element set) x) )

(define (elements set)
  ((operate 'elements set)) )
```

mp.sets.fw.scm

```
;  Message-passing implementation of a set of integers as a simple list.

(define (new-list-set element-list)

   (define (respond-to-msg msg)
      (cond
         ((equal? msg 'member?) member?)
         ((equal? msg 'with-element) with-element)
         ((equal? msg 'elements) (lambda ( ) element-list))
         (else (error "list set: unknown msg")) ) )

   (define (member? x)
      (member x element-list) )

   (define (with-element x)
      (if (member? x)
         respond-to-msg
         (new-list-set (cons x element-list)) ) )

   (define (elements)
      element-list)

   respond-to-msg)

;  Message-passing implementation of a set of integers as a sorted list of intervals.

(define (new-intvls-set element-list)
  ... )

;  Generic operations on sets.

(define (member? x set)
  ((set 'member?) x) )

(define (with-element x set)
  ((set 'with-element) x) )

(define (elements set)
  ((set 'elements)) )
```

Quiz—Data representation mechanisms

Questions on this quiz concern the mechanisms for implementing data-directed programming and message passing. They focus in particular on the use of functions that return functions, and tables that store functions.

Readings

Concrete Abstractions, sections 6.3 and 6.5; chapter 9.

Structure and Interpretation of Computer Programs, chapter 2, especially sections 2.2.4, 2.4 and 2.5.

Suggested exercises

Concrete Abstractions: exercises 6.19, 6.24, 9.3, 9.4, 9.5, 9.8, 9.11, 9.13, 9.14 through 9.18, 9.21, and 9.23.

Structure and Interpretation of Computer Programs: exercises 2.4, 2.6, 2.73, 2.74, 2.75, 2.77, 2.78, 2.79, and 2.80.

Sample questions for the "Data representation mechanisms" quiz

1. A playing card has a rank and a suit. Provide code that represents a card as a *function*. Specifically, fill in the blanks in the following framework.

```
; Return the empty pair.
(define (new-card rank suit)
   (lambda ( ___ ) ___ ) )

; Return the rank of the card card.
(define (rank card)
   ___ )

; Return the suit of the card card.
(define (suit card)
   ___ )
```

2. Suppose we wanted to provide only an integer card-number between 0 and 51 as argument to the new-card function. We would then compute the rank and suit from card-number using the expressions

   ```
   (+ 1 (remainder card-number 13))
   ```

   for the rank and

   ```
   (list-ref '(club diamond heart spade) (remainder card-number 4))
   ```

   for the suit. Modify your new-card function from exercise 1 so that it takes only a single card-number argument, and moreover it does the rank and suit computations only once per card rather than every time the rank or suit function is requested from the card.

3. Given below is a version of the code from *Concrete Abstractions* that implements data-directed programming without error checking.

```
(define (make-type name operation-table) (cons name operation-table) )

(define type-name car)
(define type-operation-table cdr)

(define (operate operation-name value)
   (table-find
      (type-operation-table (type value))
      operation-name
      (lambda (f) (f (contents value))) )

; Make a table that pairs each key with the corresponding value.
(define (make-table keys values) ... )

; Find the value in the table that corresponds to the given key, and apply f to it.
(define (table-find table key f) ... )

(define (tagged-datum type value) (cons type value) )

(define type car)
(define contents cdr)
```

Suppose now that data-directed programming were to be used with a number of types of cards, including those from a Bridge deck. Consequently, a bridge-card-type is set up with a call to make-type:

```
(define bridge-card-type
   (make-type 'bridge (make-table '(rank suit) _____ ) ) )
```

A new bridge card is then created by calling new-bridge-card:

```
(define (new-bridge-card n)
   (tagged-datum
      bridge-card-type
      (list
         (+ 1 (remainder n 13))
         (list-ref '(club diamond heart spade) (remainder n 4)) ) ) )
```

Finally, the rank of a card would be accessed via the expression

```
(operate 'rank card)
```

Which of the following would be an appropriate second argument to make-table in the bridge-card-type definition above? Briefly explain.

a. '(car cadr)

b. (list car cadr)

c. (list cadr caddr)

d. (list
      (lambda (n) (+ 1 (remainder n 13)))
      (lambda (n) (list-ref '(club diamond heart spade) (remainder n 4))))

4. Suppose that the expression (list rank suit) was supplied as the second argument to make-table in the bridge-card-type definition of exercise 3. Define the rank function.

Answers to sample questions for the “Data representation mechanisms” quiz

1. This solution is essentially a message-passing implementation of cards.

```
(define (new-card rank suit)
   (lambda (cmd)
      (cond
         ((equal? cmd 'rank) rank)
         ((equal? cmd 'suit) suit)
         (else (error "undefined card operation")) ) ) )

(define (rank card)
   (card 'rank))

(define (suit card)
   (card 'suit) )
```

2.

```
(define (new-card card-number)
   (let ((myRank (+ 1 (remainder card-number 13)))
         (mySuit (list-ref '(club diamond heart spade)
            (remainder card-number 4)) ) )
      (lambda (cmd)
         (cond
            ((equal? cmd 'rank) myRank)
            ((equal? cmd 'suit) mySuit)
            (else (error "undefined card operation")) ) ) ) )
```

3. The answer is b.

   The list of values passed as a second argument to make-table must be a list of functions. Choice a is a list of *symbols* (which coincidentally happen to be names of functions) and thus is inappropriate.

   Moreover, each function in the table is applied to (contents value), where value is the card passed to operate. That rules out choice d, in which each function is applied to an integer.

   Finally, even though a typed card is represented internally as a three-element list whose elements are the type, the rank, and the suit, the function from the table is applied to (contents value), which strips off the type before the function application. The functions in choice c would work if the type were not removed before they were applied, but they don’t work here.

4. A solution is below. Note that it shouldn’t call operate; that would lead to a mutual recursion, since operate calls the function in the table.

```
(define (rank card)
   (car (contents card)) )
```

Program—Iterators and lazy evaluation

Goals

This programming assignment gives you practice working with a Scheme data structure called a *stream*. Abstractly, a stream is just a sequence. However, it is implemented using the mechanism of *lazy evaluation*, in which an object is evaluated only at the time when, and to the extent that, it is needed. A stream may thus be used to represent an infinite sequence of elements!

A stream is also a good way to implement an *iterator*, an enumeration of elements of a collection. Iterators provide standard interfaces to collections: rather than provide numerous special-purpose functions that process the elements in various ways, one may instead provide an iterator that allows the user to determine how to process the elements.

Related quizzes

Streams.

Readings

Concrete Abstractions does not cover this topic except in passing in exercise 9.6.

Structure and Interpretation of Computer Programs, sections 1.1.5, 2.2.3, 3.5, 4.2, and 4.2.1.

The section “Stream implementation” in this document.

Problem

This assignment is described on the following pages.

Iterators and lazy evaluation

This is a three-part assignment. Don't use set! or set-car! or set-cdr! in any of the parts.

Part 1

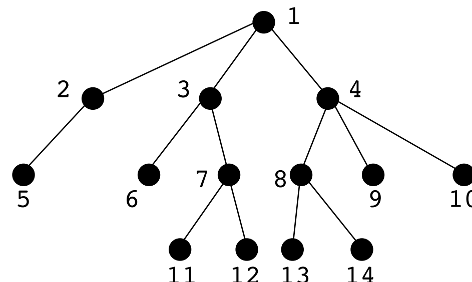
Write two functions to manipulate nonnegative proper fractions. The first function, fract-stream, will take as its argument a list of two nonnegative integers, the numerator and the denominator, in which the numerator is less than the denominator. It will return an infinite stream of decimal digits representing the decimal expansion of the fraction. The second function, approximation, will take two arguments: a fraction stream and a nonnegative integer num-digits. It will return a list (not a stream) that contains the first num-digits digits of the decimal expansion. Test your functions on the examples below.

> (fract-stream '(1 7)) should return the stream representing the decimal expansion of $\frac{1}{7}$, which is 0.142857142857142857…
> (stream-car (fract-stream '(1 7)) should return 1.
> (stream-car (stream-cdr (stream-cdr (fract-stream '(1 7))))) should return 2.
> (approximation (fract-stream '(1 7)) 4) should return (1 4 2 8).
> (approximation (fract-stream '(1 2)) 4) should return (5 0 0 0).

Part 2

Consider a tree represented as a Scheme list as follows. The root element is the first element of the list; each of the root's subtrees is a subsequent element of the list. An example tree and its representation appears below.

```
(1
  (2 (5))
  (3 (6) (7 (11) (12)))
  (4 (8 (13) (14)) (9) (10)))
```



Write two iterator functions, each of which takes a tree in the format just described as an argument and returns a stream of the tree's elements. One of the functions, depth-first-iterator, returns the elements in postorder (first the elements of subtrees, then the root). For the tree in the diagram, the stream should contain the elements (5 2 6 11 12 7 3 13 14 8 9 10). The other function, breadth-first-iterator, should return the elements in breadth-first order: first the root, then its children, then its grandchildren, and so on. For the tree in the diagram, the breadth-first stream should contain (1 2 3 4 5 6 7 8 9 10 11 12). (For more information on these traversal techniques, consult a data structures book.)

Hint: stream-append (page 340 in Abelson and Sussman) will help you simplify your solutions.

Your functions should create the stream they return without preprocessing the entire tree (e.g. you're not allowed to flatten the tree to create the breadth-first stream). Test your functions on the tree in the diagram.

Part 3

Do exercise 3.67 in Abelson and Sussman: Modify the pairs procedure (given below) so that (pairs integers integers) will produce the stream of *all* pairs of integers (i j) without the condition i ≤ j.

```
(define (pairs stream1 stream2)
   (cons-stream
      (list (stream-car stream1) (stream-car stream2))
      (interleave
         (stream-map
            (lambda (x) (list (stream-car stream1) x))
            (stream-cdr stream2) )
         (pairs (stream-cdr stream1) (stream-cdr stream2)) ) ) )
```

The functions interleave and stream-map and the variable integers are described in Abelson and Sussman. Provide a list of the first twenty or so elements of the stream when you bring your solution to a tutor for grading.

Checklist

All parts implemented as specified:
- fract-stream and approximation functions;
- streams that implement depth-first and breadth-first iteration, and that don't involve preprocessing of the argument tree;
- modification of the pairs function as specified.

Transcripts of test results.

A 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—Streams

This quiz focuses on the implementation of Scheme streams using `force` and `delay`. You should be able to identify the list structure representing a stream and to distinguish this from a list with the same elements. You should also be able to trace through a function that uses streams, identify possibilities for infinite recursion, and be aware of what problems would arise from attempting to implement streams without using special forms.

You do not need to understand the memoized version of `delay` described on page 324 of *Structure and Interpretation of Computer Programs*.

Readings

Concrete Abstractions does not cover this topic except in passing in exercise 9.6.

Structure and Interpretation of Computer Programs, sections 1.1.5, 2.2.3, 3.5, 4.2, and 4.2.1.

The section "Stream implementation" in this document.

Suggested exercises

Structure and Interpretation of Computer Programs: exercises 3.53 through 3.56 and 3.58 through 3.61.

All exercises in "Stream implementation".

Sample questions for the “Streams” quiz

1. You type

```
(delay (/ 1 0))
```

to the Scheme interpreter. What happens?

2. You type

```
(define ones (cons-stream 1 ones))
```

immediately after entering the Scheme interpreter. Why doesn’t an infinite recursion result? Why doesn’t a 1-element stream result?

3. Consider the stream function that’s analogous to map:

```
(define (stream-map f s)
   (if (stream-null? s)
      the-empty-stream
      (cons-stream (f (stream-car s)) (stream-map f (stream-cdr s))) ) )
```

Give the number of calls to force and delay that are generated by evaluation of the expression

```
(stream-map (lambda (x) (+ 1 x)) ones)
```

where ones is defined as in exercise 2.

4. Provide good comments for the istream, lstream, and mystery functions describing their arguments and the values they return.

```
; x is an integer.
(define (istream x)
   (cons-stream x (istream (+ x 1))))

(define (lstream s n)
   (if (= n 0)
      '( )
      (cons (stream-car s) (lstream (stream-cdr s) (- n 1))) ) )

(define (mystery s1 s2)
   (cons-stream (stream-car s2) (mystery (stream-cdr s2) s1)) )
```

Answers to sample questions for the "Streams" quiz

1. stk produced the following response:

```
STk> (delay (/ 1 0))
#[promise 816f358 (not forced)]
```

   Delay is a special form that does not evaluate its argument, so no crash resulted from the attempt to divide by 0.

2. ones is just a variable. If cons were substituted for cons-stream in the expression, ones would be evaluated and found to be undefined.

   The value assigned to ones is the infinite stream of 1's. This results from cons-stream delaying the evaluation of its argument; when the delayed argument is forced, ones is evaluated and found to have a 1 as a first element and a promise to evaluate ones as its "tail".

3. The call to stream-map expands to the following expression:

```
(cons-stream
   ((lambda (x) (+ 1 x)) (stream-car ones))
   (stream-map f (stream-cdr ones)) )
```

   Cons-stream evaluates its first argument but delays evaluation of its second (and thus doesn't call stream-cdr). Stream-car involves neither a force nor a delay. Thus one call to delay and no calls to force result from the specified evaluation.

4. Given an integer x, istream returns the stream of integers x, x+1, x+2, … Given a stream s and a nonnegative integer n that's less than or equal to the length of s, lstream returns a list of the first n elements of s. Given two streams s1 and s2, mystery returns the interleaved elements of the streams, starting with the first element of s2, then the first of s1, then the second of s2, and so on.

Rule-based programming

Definitions

A rule-based programming environment consists of

- A *data base* (which we'll leave undefined for now);
- A set of *rules*: condition-action pairs. The condition evaluates some aspect of the data base. The action(s) typically modify some aspect of the data base.
- A *monitor* that determines which rules can be applied (i.e. which rules have conditions that evaluate to true), and that chooses an applicable rule and applies it.

What happens in a rule-based environment

The following sequence of events is repeated by the monitor:

1. Decide what rules are applicable.
2. Choose a rule to apply. When there are more than one applicable rule, there are several possible strategies ("meta-rules"):
   a. choose the highest priority rule;
   b. choose the most specific rule;
   c. choose the rule that refers to the element most recently added to the data base;
   d. choose the most recently applied rule;
   e. choose the rule that has been applicable for the longest time;
   f. choose a rule that has not previously been applied;
   g. choose the first rule in the list;
   h. make a random choice.

   The monitor can also choose not to choose, i.e. to explore all the applicable rules in parallel.
3. Apply the rule.

How this is done in Scheme

Here is a function that implements the monitor of a rule-based system. It takes a list of rules and a data base as arguments.

```
(define (monitor rules data-base)
   (let ((rule-set (applicable rules database)))
      (if (not (empty? rule-set))
         (monitor rules
            (apply-rule (choose rule-set) data-base) ) ) ) )
```

The `applicable` function returns a list of applicable rules; it often involves pattern-matching. The `choose` function applies one of the criteria mentioned above. (Note the difference between this and the way a Scheme `cond` is executed.) The `apply-rule` function "executes" the actions in the action-part of the rule.

Applications

Rule-based programming is good for *expert systems*, in which expertise is modelled by rules. The expertise is typically difficult to organize into a sequential set of conditions to be checked. Some categories:

category	*problem addressed*
interpretation	inferring situation descriptions from sensor data
prediction	inferring likely consequences of given situations
diagnosis	inferring system malfunctions from observables
design	configuring objects under constraints
planning	designing actions
monitoring	comparing observations to plan vulnerabilities
debugging	prescribing remedies for malfunctions
repair	executing a plan to administer a prescribed remedy
instruction	diagnosing, debugging, and repairing student behavior
control	interpreting, predicting, repairing, and monitoring system behaviors

Note an advantage of rule-based programming: one can ask why a given conclusion was reached.

References

Building Expert Systems, F. Hayes-Roth et al. (editors), Addison-Wesley, 1983.

Handbook of Artificial Intelligence, A. Barr et al. (editors), Addison-Wesley.

Stream implementation

Functional programming relies on the ability to pass and return collections of data to and from functions. But what if the collection is infinite or indefinitely long, for instance, the collection of user responses in a run of the Doctor program? Such a collection is called a *stream*. How might such a thing be implemented?

Here's a puzzle. Find the next number in the following sequence:

1, 4, 9, 16, 25, …

It's likely that you got the answer by figuring out the algorithm—i.e., the function—for producing the members of the sequence. Thus you store the sequence, the stream of squares of integers, in a form something like the following:

```
(1   4   9   16   25   something involving square)
```

In Scheme, one may store a stream as a pair based on the principle we just noticed. The car of the pair will be the first element of the stream. The cdr will be some way to produce the rest of the stream. We will use the function cons-stream to build a stream, functions stream-car and stream-cdr to access the parts of a stream built with cons-stream, and stream-null? to check if a stream is empty. The symbol the-empty-stream will represent the empty stream.

The implementation of streams is described in Abelson and Sussman:

> "Our implementation of streams will be based on a special form called delay. Evaluating (delay *exp*) does not evaluate the expression *exp*, but rather returns a so-called delayed object, which we can think of as a "promise" to evaluate *exp* at some future time. As a companion to delay, there is a procedure called force that takes a delayed object as argument and performs the evaluation—in effect, forcing the delay to fulfill its promise."

Delay can package its argument by treating it as the body of a function; (delay *exp*) is implemented as

```
(lambda ( ) exp)
```

Force simply calls the packaged procedure:

```
(define (force delayed-object) (delayed-object))
```

Here are the implementations of stream-car, stream-cdr, and cons-stream.

```
(define (stream-car stream) (car stream))

(define (stream-cdr stream) (force (cdr stream)))

(define (cons-stream a stream) (cons a (delay stream)))
```

A predefined variable the-empty-stream represents the empty stream, and is identified via the stream-null? function:

```
(define (stream-null? stream) (equal? stream the-empty-stream))
```

Here's an example that uses these functions to define the stream of all integers greater than or equal to a particular integer:

```
(define (integers k)
   (cons-stream k (integers (+ k 1)) )
```

To access elements of the stream, we do the following:

```
=> (stream-car (integers 1))
1
=> (stream-car (stream-cdr (integers 1)))
2
=> (stream-car (stream-cdr (stream-cdr (integers 1))))
3
```

Stream exercises

1. What should (delay (+ 1 27)) return? What about (force (delay (+ 1 27)))?
2. What would be wrong with the construction

```
(stream-cdr (stream-cdr (cons-stream 1 '(2 3)))?
```

Consider the following two definitions:

```
(define (list-interval low high)
   (if (> low high) '( )
      (cons low (list-interval (+ low 1) high)) ) )

(define (stream-interval low high)
   (if (> low high) the-empty-stream
      (cons-stream
         low
         (stream-interval (+ low 1) high) ) ) )
```

3. What should (delay (list-interval 1 3)) return? What's the difference between this and (stream-interval 1 3)?
4. Does (stream-car (stream-interval 1 3)) return the same result as (car (list-interval 1 3))? Why or why not?
5. Does (stream-cdr (stream-interval 1 3)) return the same result as (cdr (list-interval 1 3))? Why or why not?