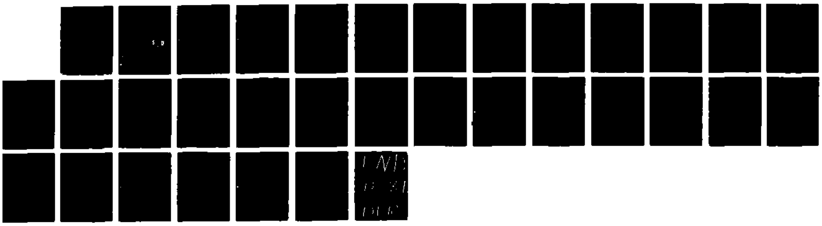
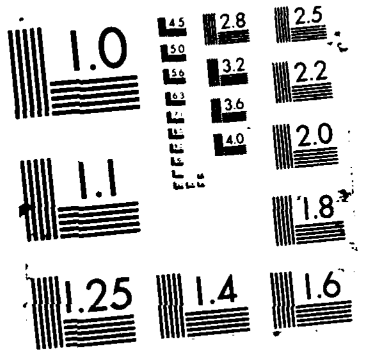
AD-A185 606   LISP: A LANGUAGE FOR STRATIFIED DESIGN(U) MASSACHUSETTS   1/1
INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB
H ABELSON ET AL. AUG 87 AI-M-986 N00014-86-K-0180

UNCLASSIFIED   F/G 12/5   NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(12)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 986	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Lisp: A Language for Stratified Design		5. TYPE OF REPORT & PERIOD COVERED AI-Memo
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Harold Abelson and Gerald Jay Sussman		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE August 1987
		13. NUMBER OF PAGES 31
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Abstraction, scheme, stratified design, lisp

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We exhibit programs that illustrate the power of Lisp as a language for expressing the design and organization of computational systems. The examples are chosen to highlight the importance of abstraction in program design and to draw attention to the use of procedures to express abstractions.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 986 *August 1987*

Lisp: A Language for Stratified Design

Harold Abelson and Gerald Jay Sussman

Abstract

We exhibit programs that illustrate the power of Lisp as a language for expressing the design and organization of computational systems. The examples are chosen to highlight the importance of *abstraction* in program design and to draw attention to the use of procedures to express abstractions.

Keywords: Abstraction, scheme, stratified design, Lisp.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-86-K-0180.

Lisp: A language for stratified design

Harold Abelson and Gerald Jay Sussman

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

August 1987

Abstract

We exhibit programs that illustrate the power of Lisp as a language for expressing the design and organization of computational systems. The examples are chosen to highlight the importance of ***abstraction*** in program design and to draw attention to the use of procedures to express abstractions.

Just as everyday thoughts are expressed in natural language, and formal deductions are expressed in mathematical language, methodological thoughts are expressed in programming languages. A programming language is a medium for communicating methods, not just a means for getting a computer to perform operations—programs are written for people to read as much as they are written for machines to execute.

This article exhibits programs that illustrate the power of Lisp as a language for expressing the design and organization of computational systems. The examples are chosen to highlight the importance of ***abstraction*** in program design and to draw attention to the use of procedures to express abstractions. Any programming language provides primitive components, means by which these can be combined, and means by which patterns of combination can be named and manipulated as if they were primitive. With appropriate abstractions to separate the specification of components from the details of their implementation we can provide a library of standard components that can be freely interconnected, allowing great flexibility in design.

A language for design should not unnecessarily limit our ability to make abstractions. Most traditional programming languages, however, place arbitrary restrictions on procedural abstractions. Three common restrictions are: (1) requiring that a procedure be named and then referred to by name, rather than stating its definition at the point of reference; (2) forbidding procedures to be returned as the values of other procedures; (3) forbidding procedures to be components of such data structures as records or arrays.

The well-publicized programming methodology of top-down, structured design produces systems that are organized as trees. Following this methodology, a system is designed as a predetermined combination of parts that have been carefully specified to be combined as determined. Each of the parts is itself designed separately by this same process. The methodology is flawed: if a system is to be robust it must have more generality than is needed for the particular application. The means for combining the parts must allow for after-the-fact changes in the design plan as bugs are discovered and as requirements change. It must be easy to substitute parts for one another and to vary the arrangement by which parts are combined. This is necessary so that small changes in the problem to be solved can be effected by making small changes in the design.

To this end expert engineers stratify complex designs. Each level is constructed as a stylized combination of interchangeable parts that are regarded as primitive at that level. The parts constructed at each level are used as primitives at the next level. Each level of a stratified design may be thought of as a specialized language with a variety of primitives and means of combination appropriate to that level of detail. For example, in electrical design, resistors and transistors are combined as analog circuits to make TTL, a language appropriate to digital circuits. TTL parts are in turn combined to build processors, bus structures, and memory systems appropriate to computer architecture. The real power of Lisp is that its unrestricted abstractions support the construction of new languages, greatly facilitating the strategy of stratified design.

The programs in this article are written in the Scheme dialect of Lisp. Scheme is an especially good vehicle for exhibiting the power of procedural abstractions because, to a greater extent than other Lisp dialects, Scheme does not distinguish between patterns that abstract over procedures and patterns that abstract over other kinds of data.

1 Expressing abstractions as procedures

Procedural abstractions can help elucidate a process by allowing us to express it as an instance of a more general idea. Consider the simple square root program of figure 1. The algorithm here is implemented in a straightforward way—the internal procedure **try** is iterated to repeatedly improve a guess for the square root until the guess is good enough.

Although the square-root implementation is straightforward, it does not express the underlying idea in generalizable form. It is not built out of components that can be easily isolated for use in solving other problems. A clearer way to formulate the algorithm is as a process of computing a *fixed point*: the square root of a radicand x is the number y such that $y = x/y$ or, in other words, y is a fixed point of the procedure

```
(lambda (y) (/ x y))
```

How do we find a fixed point of a function? In favorable cases, we can iterate the function until the result is close to the input. For example, during boring meetings many of us have noticed that we can find the fixed-point of the cosine function by entering 1 on a pocket calculator and repeatedly pressing the cosine button. After a while the calculated value converges to approximately .739085. We can capture that general idea as the **fixed-point** procedure shown in figure 2. **Fixed-point**, given a one-argument procedure **f** and an initial value, keeps applying **f** until successive values are close to each other.

We can attempt to find square roots by

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y)) 1))
```

Unfortunately, this doesn't work. Unlike the cosine function, applying the indicated procedure over and over does not converge to a fixed point, but rather alternates between the same two values, which are on opposite sides of the square root.

In situations like this we can often force convergence by averaging. The **average-damp** procedure shown in figure 3 takes as its argument a procedure that computes a function f and returns as its result a procedure that computes a function with the same fixed point as f, but whose oscillations are damped out by averaging successive values. Figure 3 also shows how to use **average-damp** to express the square root method as a process of finding the fixed point of an average-damped function.

The advantage of this formulation is that it decomposes the method into useful pieces—finding fixed-points of general functions and using damping to encourage convergence. These ideas are formalized as procedural abstractions, identifiable units that are available to be used in other contexts.

2 Stratified design

Peter Henderson [5] used stratified design in a beautiful analysis of the construction of the "Square Limit" woodcut of M. C. Escher. He created a sequence of languages that makes it easy to describe such images. There is a language of primitive pictures that are constructed from points and lines. Built on top of this is a language of geometric combination that describes how pictures are placed relative to one another to make compound pictures. Built on this is a language that abstracts common patterns of picture combination.

In Henderson's system, a picture is represented by a picture-drawing procedure that takes a rectangle and draws an image scaled to fit the rectangle. At the lowest level of description, a picture-drawing procedure can be generated beginning with a collection of geometric elements specified in terms of (x, y) coordinates with respect to the unit square $(0 \leq x \leq 1; 0 \leq y \leq 1)$. Figure 4 shows two simple pictures, **diamond** and **leg**, each constructed from a set of line segments by the procedure **primitive-picture**. Besides segments, other picture elements appropriate to this level are circles with specified radii and centers, spline curves through designated points, and so on.

The **primitive-picture** procedure itself (figure 5) takes a list of line segments and returns the corresponding picture-drawing procedure. For any rectangle, the scaling and shifting required to transform geometric elements to fit the rectangle can be described by an affine transformation on points in the plane that maps the unit square to the rectangle. The **point-map** procedure shown in figure 6 takes a rectangle and returns the appropriate transformation for that rectangle.

The next level of description in Henderson's system is a language of geometric combinators that place pictures beside or above one another and rotate pictures through multiples of 90 degrees. For instance, the **beside** combinator illustrated in figure 7 adjoins two pictures horizontally so that their widths are in a given ratio.

One important feature of Henderson's geometric combinators is that the set of all pictures is *closed* under combination: The `beside` of two pictures is itself a picture and can therefore be further combined with other pictures. In addition, combinators can be abstracted: We can express common patterns of picture combination as new picture combinators defined in terms of other combinators. For example, a `triangle` combination is formed by placing one picture above two copies of another as shown in figure 8. Since combinators are expressed procedurally, we have all the power of Lisp at our disposal in defining complex combinators. Figure 9 shows the recursive combinator `right-push`. The combinator language derives its power from the closure and abstraction properties of the combinators. That is why it can describe seemingly complex figures using only a few simple ideas.

The combinators themselves are manipulated at a third linguistic level that describes common patterns of combining picture combinators. Just as the square root algorithm above is elucidated by expressing it as a fixed-point computation, `right-push` can be re-expressed as an instance of a general pattern of "pushing"—repeatedly applying a combinator:

```
(define right-push (push beside))
```

Figure 10 shows how to define `push` as a procedure that transforms combinators to combinators. Having isolated the `push` abstraction, we can apply it to other combinators such as `triangle`, and we can use the resulting derived combinators to produce simple, stratified descriptions of complex pictures, such as the one in figure 10.

The stratified description of the picture in figure 10 is flexible. We can vary the pieces at any level: We can change the location of a point in the primitive picture `leg`, we can replace the compound picture `animal` by some other basic repeated unit, we can replace `triangle` by some other combintor to be pushed, or we can replace `push` by some other transformation of combinators.

3 Metalinguistic abstractions

Procedural abstractions are a source of power in creating stratified designs—we build structures by composing procedures, we abstract common patterns of usage, and we build upon this framework. But for some problems the appropriate means of combination may be awkward to express as compositions of procedures; towers of abstractions may not suffice.

The natural programming style for Lisp is functional—the structural units are single-valued functions implemented by procedures. Within this style, Lisp accommodates object-oriented programming and imperative programming. Traditional algorithmic languages such as Pascal, C, and FORTRAN are more naturally imperative—the statements and the subroutines we build modify the memory of an abstract machine. In a logic programming language such as Prolog the natural structural units are (multi-valued) relations rather than (single-valued) functions or imperative operations. In simulation or artificial-intelligence applications it is natural to describe processes in event-driven style by specifying collections of rules that correspond to conditions or goals. Each of these programming paradigms is legitimate, but no single paradigm is sufficient—large systems typically have some parts that are naturally described using one style and other parts that are more naturally expressed in other ways.

Part of the wonder of computation is that we have the freedom to change the framework by which the descriptions of processes are combined. If we can precisely describe a system in *any* well-defined notation, then we can build an interpreter to execute programs expressed in the new notation, or we can build a compiler to translate programs expressed in the new notation into any other programming language.

When we design a system that incorporates an application language, we must address *metalinguistic* issues as well as linguistic issues. That is to say, we must consider not only how to describe a process, but also how to describe the language in which the process is to be described. We must view our programs from two perspectives: From the perspective of the interpreter or compiler, an application program is merely data, and the interpreter or compiler operates on that data without reference to what the program is intended to express. When we write programs in the application language, we view that same data as a program with meaning in the application domain.

Interpreters and compilers are just programs, but they are special programs. An application program to be interpreted or compiled is not a composition of abstractions of the language in which the interpreter is implemented or to which the compiler translates. This linguistic shift transcends the limits of abstraction.

One can implement interpreters and compilers in any programming language, but Lisp's facility with symbolic data provides unusually good support for developing such subsystems. The Lisp community regards metalinguistic abstraction as a standard programming technique. Almost every large Lisp program includes interpreters and compilers for several specialized languages, each tailored to a specific part of the application problem.

3.1 A rule language

Consider the problem of simplifying algebraic expressions. For example, it should be possible to simplify

$$(x + \sin(xy))\,(\sin(xy) - x) + \cos(xy)\cos(yx)$$

to $1 - x^2$. The simplification process may be captured as a collection of rules and a strategy for applying them. The rules embody the commutative and distributive laws, various trigonometric identities, and so on. Each rule describes how to reduce expressions of a certain form to simpler equivalent expressions. Given an expression, one finds an applicable rule, transforms the expression accordingly, and then attempts to simplify the transformed expression. The process continues until one reaches an expression to which no rules apply.

Figure 11 shows some algebraic-simplification rules written in a language that was designed specifically for implementing simplifiers. A rule in the language has three parts: a *pattern*, some extra conditions, and a *skeleton*. The pattern specifies the class of expressions to which the rule is applicable. Patterns may contain constant terms and pattern variables that indicate either individual elements or arbitrary *segments*—sequences of zero or more elements—that can appear at designated positions in the expression. One can also use *pattern-variable predicates* to restrict the range of values of an element or segment. A rule is applicable to an expression if the rule pattern matches a subexpression and if the extra conditions are satisfied. The result of a rule application is the original expression with the matched subexpression replaced by the instantiated rule skeleton. The instantiated skeleton is formed by substituting values provided by the match in place of the indicated skeleton variables.

In all, about 30 rules such as the ones in figure 11 suffice to produce an algebraic simplifier that operates on expressions represented using Lisp-style prefix notation. Having created the rule language, we can apply it to other problems as well by specifying other sets of rules. For example, we could do algebraic simplification using some other syntax for algebraic expressions, or we could build a peephole optimizer for a compiler by specifying a collection of rules that reduce specified sequences of computer instructions to more efficient, equivalent sequences.

3.2 The rule interpreter

Programming a simplifier in the rule language allows us to focus on the rules themselves, without becoming distracted by concerns about the strategy and mechanism by which rules are applied. These issues are faced by the rule interpreter.

Given a list of rules, **`make-simplifier`**, shown in figure 12, returns a simplification procedure that applies these rules. This procedure embodies a rule-application strategy whereby rules are repeatedly applied until the expression is unchanged. Using this strategy with a set of arbitrary rules can be dangerous—the rule application process may not terminate, or the result of applying the rules may not be well-defined. With this strategy, we must be careful to propose rules that yield a *reduction* process—one that will eventually terminate with a *canonical form*.

The mechanism of rule application is implemented by **`try-rules`**, which is shown in figure 13. The pattern-matcher used by **`try-rules`** (see figures 14, 15, and 16) is written in *continuation-passing style*. In this style, a procedure is passed continuation procedures that are to be called when the procedure is done. Continuation-passing style can be used to implement many sophisticated program control structures. The matcher exploits continuation style to implement the *backtracking* required to handle segment variables.

The difficulty with segment variables is that they allow a pattern to match a given data item in more than one way. For example, the pattern

```
((?? x) (?? y) (?? x) (?? z))
```

can be matched against the expression

```
(1 2 3 1 2 3 1 2 3)
```

to give the dictionary

```
x : (1)
y : (2 3)
z : (2 3 1 2 3)
```

or the dictionary

```
x : (1 2)
y : (3)
z : (3 1 2 3)
```

or the dictionary

```
x : (1 2 3)
y : ( )
z : (1 2 3)
```

or 13 other possible dictionaries.

One way to handle such multiple matches is to design the matcher so that it returns a list of all possible dictionaries that could complete the match. But this would be very inefficient—for a complex pattern there could easily be thousands of possibilities and there is no need to generate them all. An alternative idea is to have the matcher generate a single dictionary to be used by the rule interpreter, *together with* a way to go back and generate more possibilities if the first one proves to be unsuitable, for example, if the values in the dictionary turn out not to satisfy the rule predicate. The strategy of returning to a previous choice point in a program to try more possibilities is called *backtracking*. While there are many ways to implement backtracking, continuation passing fills the need nicely, because the "place to go back to" can be embedded in the failure continuation.

Figure 16 shows the procedure `segment-match`, which uses continuations to implement backtracking in this way. Where there is no value for the pattern variable already in the dictionary, the matcher is free to bind the segment variable to *any* initial segment of the datum. The choice is made using the internal procedure `try-segment`, whose argument is the rest of the data after the segment to be tried. The success continuation for this choice is the same as for the original call. The failure continuation reruns `try-segment` choosing a longer initial segment. Successive failures will try longer and longer segments until either the match succeeds or the data runs out and the original `fail` continuation is invoked.

3.3 Memoizing to avoiding redundant computation

If a subexpression appears in an expression more than once, the rule-interpreter will go through the work of simplifying the expression each time it appears. For example, in simplifying `(+ (* x 2 y) (* x 2 y))`, the subexpression `(* x 2 y)` is simplified twice. Such redundancy is typical of programs that use recursive decomposition. Perhaps the best-known example of this phenomenon is in computing the Fibonacci numbers using the recurrence relation:

$$\mathrm{Fib}(0) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{if } n \geq 2 \end{cases}$$

Interpreting this recurrence as a recursive procedure leads to a process in which the time required to compute Fib(n) grows exponentially with n, due to the redundant computations.

One could implement another algorithm for computing Fibonacci numbers, or make *ad hoc* modifications to the rule interpreter's control structure. However, there is a technique for dealing with this problem in a uniform way by using *memoized procedures*. A memoized procedure maintains a history of the values it has been asked to compute. When asked to compute a value, the procedure first checks to see if that value has already been computed. If so, it returns the remembered value immediately. If not, it computes the value and records this for future reference. Figure 17 shows a higher-order procedure `memoize`, which, given a procedure `f` as argument, returns a memoized version of `f`. The memoized version of `f` is called in the same way that `f` is, but will run be much faster because it avoids redundant computation. Figure 18 shows the `make-simplifier` procedure redefined using the memoizer.

Like the `fixed-point` procedure discussed earlier, **`memoize`** expresses a general method of computing as a higher-order procedure. Also, as with `fixed-point`, the power of this approach is that we can divide programs into meaningful modules, thus separating the particular task of simplifying expressions or computing Fibonacci numbers from the general strategy of memoizing.

4 Conclusion

People who first learn about Lisp often want to know for what particular programming problems Lisp is "the right language." The truth is that Lisp is not the right language for any particular problem. Rather, Lisp encourages one to attack a new problem by implementing new languages tailored to that problem. Such a language might embody an alternative computational paradigm, as in the rule language. Or it might be a collection of procedures that implement new primitives, means of combination, and means of abstraction embedded within Lisp, as in the Henderson drawing language. A linguistic approach to design is an essential aspect not only of programming but of engineering design in general. Perhaps that is why Lisp, although the second-oldest computer language in widespread use today (only FORTRAN is older), still seems new and adaptable, and continues to accommodate current ideas about programming methodology.

5 A note on Scheme

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. The Scheme dialect of Lisp demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

References

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press: Cambridge, MA, 1985.

[2] Kent Dybvig, *The Scheme Programming Language.* Prentice-Hall: Englewood Cliffs, NJ, 1987.

[3] Mike Eisenberg, *Programming in Scheme.* Scientific Press: Palo Alto, CA, 1987.

[4] Daniel P. Friedman and Matthias Felleisen, *The Little LISPer*, MIT Press, Cambridge, MA, 1987.

[5] Peter Henderson, "Functional Geometry", in *Proceedings of the 1982 ACM Symposium on on Lisp and Functional Programming.*

[6] Jonathan Rees and William Clinger (ed.), *Revised*^{3} *report on the Algorithmic Language Scheme*, ACM SIGPLAN Notices, vol. 21, no. 12, December 1986, pp. 37–79. Also available as memo. number 848a, MIT Artificial Intelligence Laboratory.

[7] Stephen Slade, *The T Programming Language: A Dialect of Lisp.* Prentice-Hall: Englewood Cliffs, NJ, 1987.

[8] William Wong, "PC Scheme: A Lexical Lisp", *Byte*, vol. 12, no. 3, March, 1987, pp. 223–226.

```
(define (sqrt x)
  (define epsilon 1.0e-10)
  (define (good-enough? guess)
    (< (abs (- (square guess) x))
       epsilon))
  (define (improve guess)
    (average guess (/ x guess)))
  (define (try guess)
    (if (good-enough? guess)
        guess
        (try (improve guess))))
  (try 1))
```

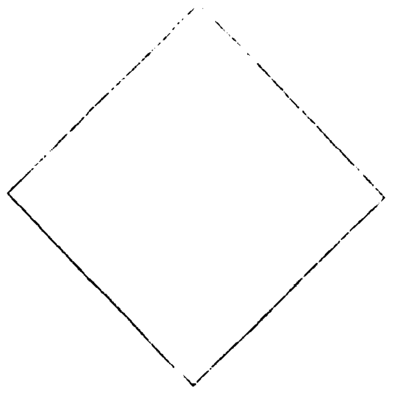
Figure 1: Scheme is a block-structured language, incorporating the internal definitions and lexical scoping of the Algol family of languages into a modern Lisp dialect. This simple Scheme procedure computes the square root of its argument using the method of successive averaging attributed to Heron of Alexandria. (Numerical analysts: Please forgive us for the end test.)

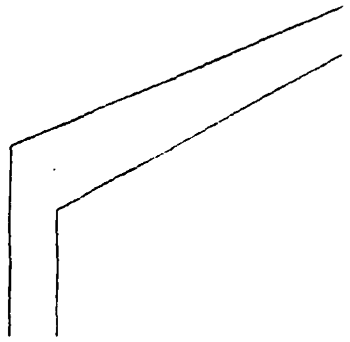
```
(define (fixed-point f initial-value)
  (define epsilon 1.0e-10)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) epsilon))
  (define (loop value)
    (let ((next-value (f value)))
      (if (close-enough? value next-value)
          next-value
          (loop next-value))))
  (loop initial-value))

(fixed-point cos 1) --> .739085
```

Figure 2: This procedure implements the "boring meeting" method of finding a fixed point of a function. Note the use of this procedure to find a fixed point of cosine.

```
(define (average-damp f)
  (lambda (x)
    (average x (f x))))

(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1))
```

Figure 3: `Average-damp` is a procedure that takes a procedure `f` as an argument and returns a procedure, the value of `(lambda (x) ...)`, as a value. Such general methods combine to allow a very clear description of Heron's algorithm.





```
(define diamond
  (let ((v1 (vertex 0.5 0))
        (v2 (vertex 1 0.5))
        (v3 (vertex 0.5 1))
        (v4 (vertex 0 0.5)))
    (primitive-picture
      (list
        (segment v1 v2)
        (segment v2 v3)
        (segment v3 v4)
        (segment v4 v1)))))
```

```
(define leg
  (let ((v1 (vertex 0.125 0))
        (v2 (vertex 0.25 0))
        (v3 (vertex 1 0.75))
        (v4 (vertex 1 0.875))
        (v5 (vertex 0.25 0.333))
        (v6 (vertex 0.125 0.5)))
    (primitive-picture
      (list
        (segment v1 v6)
        (segment v6 v4)
        (segment v3 v5)
        (segment v5 v2)))))
```

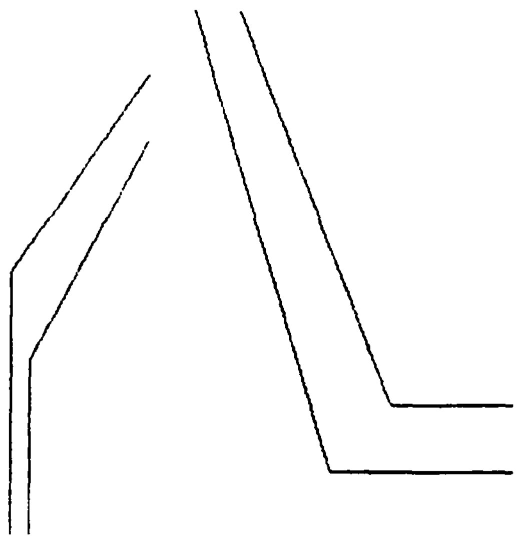
Figure 4: At the lowest level of description in Henderson's language, pictures are specified as collections of individual geometric elements. Here are two simple pictures, **diamond** and **leg**.

```
(define (primitive-picture segments)
  (lambda (rect)
    (for-each
     (lambda (segment)
       (drawline ((point-map rect) (start-point segment))
                 ((point-map rect) (end-point segment))))
     segments)))
```

Figure 5: The `primitive-picture` procedure takes a list of segments and produces a picture. A segment is a structure from which we can extract a start point and an end point. `Drawline` is a procedure that draws a line between two specified points. `For-each` takes a procedure and a list and applies the procedure consecutively to each item in the list.

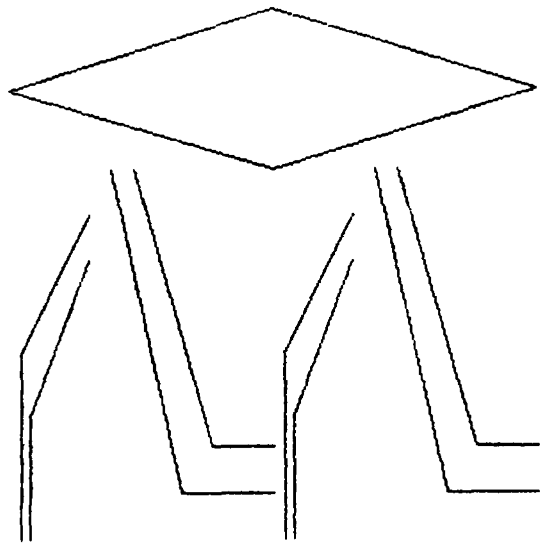
```
(define (point-map rect)
  (lambda (point)
    (+vect (scale-vector (x-coordinate point) (bottom-edge rect))
           (scale-vector (y-coordinate point) (left-edge rect))
           (origin rect))))
```

Figure 6: A rectangle is a data structure from which we can select vectors representing the bottom and left edges of the rectangle (that is, the vectors that run from the origin of the rectangle to the lower right corner and to the upper left corner) and the vector that runs from the origin of the coordinate space to the origin of the rectangle. Scheme, as a dialect of Lisp, provides list operations from which we can construct compound data objects such as rectangles and vectors, together with operations on compound data objects, such as vector addition and scaling. `Point-map` produces a transformation that, for a given rectangle, maps the interior of the unit square onto the interior of the rectangle. The transformed image of a point (x, y) is obtained by scaling by the bottom edge by x, the left edge by y, and summing these together with the origin vector.



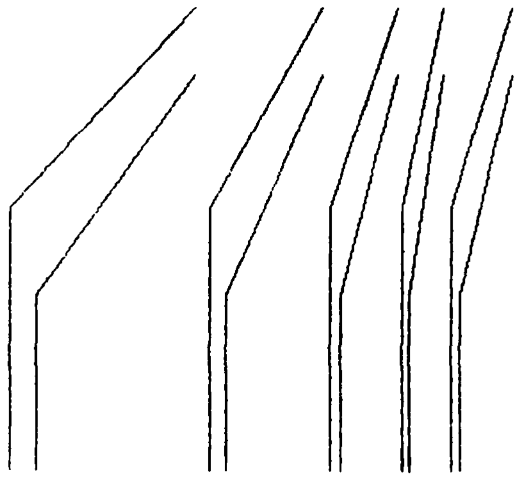
```
(define legs (beside leg (rotate90 leg) 0.3))

(define (beside pict1 pict2 ratio)
  (lambda (rect)
    (pict1 (left-subrectangle rect ratio))
    (pict2 (right-subrectangle rect ratio))))
```

Figure 7: Here is a compound picture formed from the primitive element **leg**. The picture language includes geometric combinators for adjoining pictures horizontally and vertically, and for rotating pictures by 90 degrees. **Beside** takes two pictures and a ratio and returns the procedure that, given a rectangle, splits the rectangle according to the ratio into right and left subrectangles and draws one picture in each part.



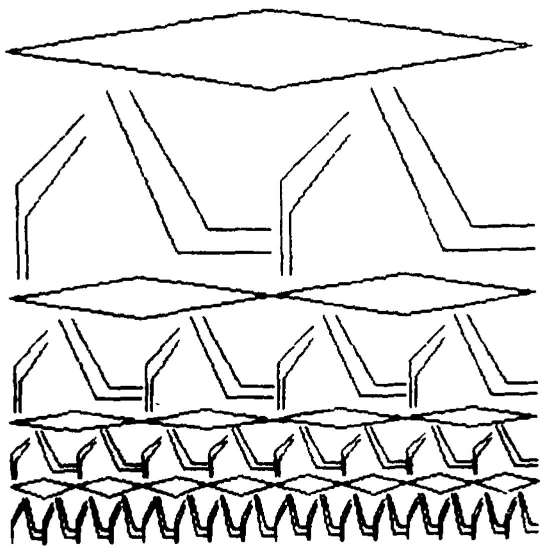
```
(define animal (triangle diamond legs 0.3))

(define (triangle pict1 pict2 ratio)
  (above pict1
         (beside pict2 pict2 0.5)
         ratio))
```

Figure 8: The **triangle** combinator places one picture above two copies of another. We can use **triangle** to combine **diamond** and **legs**.



```
(right-push leg 4 0.4)

(define (right-push pict n ratio)
  (if (= n 0)
      pict
      (beside pict
              (right-push pict (- n 1) ratio)
              ratio)))
```

Figure 9: **Right-push** is a recursively-defined combinator that repeatedly adjoin *n* copies of a picture, scaled by a given ratio. Here we see the result of adjoining **leg** to itself 4 times.



```
((push triangle) animal 3 0.5)

(define (push combiner)
  (lambda (pict n ratio)
    (define (basic-combination p)
      (combiner pict p ratio))
    ((repeated-application n basic-combination) pict)))

(define (repeated-application n operator)
  (if (= n 0)
      identity-operator
      (compose operator
               (repeated-application (- n 1)
                                     operator))))

(define (compose f g)
  (lambda (x) (f (g x))))

(define identity-operator (lambda (x) x))
```

Figure 10: **Push** is a higher-order combinator that transforms combinators to more elaborate combinators. The helper procedure **repeated-application** takes a positive integer *n* and a procedure *p* and returns the procedure that applies *p* *n* times.

```
Rule 1:  ( (- (? x) (? y))   ;pattern
           none              ;extra condition
           (+ (: x) (* -1 (: y))) )  ;skeleton

Example: (- a b) --> (+ a (* -1 b))

Rule 2:  ( (* (?? a) (+ (? b) (?? c)) (?? d))
           none
           (+ (* (:: a) (: b) (:: d)) (* (:: a) (+ (:: c)) (:: d))) )

Example: (* w x (+ p q r) z) -->  (+ (* w x p z) (* w x (+ q r) z))

Rule 3:  ( (+ (? c1 number?) (? c2 number?) (?? s3))
           none
           (+ (: (+ c1 c2)) (:: s3)) )

Example: (+ 3 4 x y) --> (+ 7 x y)

Rule 4:  ( (* (?? s1) (? f1) (?? s2) (? f2) (?? s3))
           (same-base? f1 f2)
           (* (^ (: (base f1)) (: (+ (exponent f1) (exponent f2))))
              (:: s1) (:: s2) (:: s3)))

Example: (* a (+ b c) (^ x 3) y (^ x 4) (^ z 2))
           --> (* (^ x 7) a (+ b c) y (^ z 2))
```

Figure 11: Here are some algebraic-simplification rules expressed in a reduction-rule language, together with examples of their use. Question marks in patterns denote pattern variables. The pattern in rule 1 matches any list of three elements that begins with a minus sign. The colons in skeletons indicate that the corresponding values for the pattern variables are to be substituted when the skeleton is instantiated. Double question marks, as in rule 2, denote segment variables—these can match any sequence of 0 or more elements. Double colons in skeletons specify that the segment value is to be spliced in at this position during instantiation. Rule 3 illustrates the use of predicates for pattern variables. Here `c1` and `c2` must satisfy the `number?` predicate. In general a pattern-variable predicate can be any Lisp procedure. Rule 4 uses an extra condition to specify that factors `f1` and `f2` must have the same base if their exponents are to be combined. Observe that instantiating the skeletons for rules 3 and 4 requires more than just substitution—embedded Lisp expressions must be evaluated.

```
(define (make-simplifier the-rules)
  (define (simplify-exp exp)
    (let ((result
           (try-rules (if (compound? exp)
                          (map simplify-exp exp)
                          exp)
                      the-rules)))
      (if (equal? result exp)
          result
          (simplify-exp result))))
  simplify-exp)
```

Figure 12: Given a set of rules, **make-simplifier** produces a procedure that applies these rules to expressions. It recursively simplifies every subexpression of a compound expression and simplifies the resulting combination of the simplified parts. Simplification is iterated until the expression is unchanged.

```
(define (try-rules exp the-rules)
  (define (scan rules)
    (if (null? rules)
        exp
        (match (pattern (car rules)) exp (make-empty-dictionary)
               ;; procedure to call if the match fails
               (lambda () (scan (cdr rules)))
               ;; procedure to call if the match succeeds
               (lambda (dict fail)
                 (if (check-predicate (rule-condition (car rules))
                                      dict)
                     (instantiate (skeleton (car rules)) dict)
                     (fail))))))
  (scan the-rules))
```

Figure 13: **`Try-rules`** performs a sequential scan through the rules, using a pattern matcher to check whether a rule is applicable. **`Match`** is designed to be called with five arguments—a pattern, an expression, a dictionary of bindings for pattern variables, a procedure to be executed if the match fails, and a procedure to be executed if the match succeeds. If the match is successful, **`try-rules`** checks that the values in the dictionary satisfy the rule's extra conditions. If they do, the dictionary is used to instantiate the skeleton part of the rule. This is returned as the result of the rule application. If the dictionary fails the extra condition test **`try-rules`** continues just as if the match had failed, by calling the **`fail`** procedure.

```
(define (match pat dat dict fail succeed)
  (cond ((eq? pat dat)
         (succeed dict fail))
        ((arbitrary-element? pat)
         (element-match pat dat dict fail succeed))
        ((constant? pat)
         (if (same-constant? pat dat) (succeed dict fail) (fail)))
        ((start-arbitrary-segment? pat)
         (segment-match pat dat dict fail succeed))
        ((constant? dat) (fail))
        (else
         (match (car pat) (car dat) dict
                fail
                (lambda (dict fail)
                  (match (cdr pat) (cdr dat) dict
                         fail
                         succeed))))))
```

Figure 14: **Match** is a recursive comparison that proceeds by case analysis. Pattern variables are of two classes, elements and segments, handled by **element-match** and **segment-match**. If the pattern and the datum are both general compound expressions, **match** calls itself recursively to match the first element of the pattern against the first element of the datum. The **fail** continuation for this sub-match is the original **fail** continuation. If the sub-match succeeds, the rest (cdr) of the pattern is matched against the rest of the datum using the dictionary produced by the sub-match and the continuations specified for the original match.

```
(define (element-match pat dat dict fail succeed)
  (let ((vname (var-name pat)) (p (var-restriction pat)))
    (let ((v (lookup vname dict)))
      (if (entry-exists? v)
          (let ((val (element-in v)))
            (if (and (equal? val dat)
                     (apply-restriction p val))
                (succeed dict fail)
                (fail)))
          (if (apply-restriction p dat)
              (succeed (extend-dictionary vname dat dict) fail)
              (fail))))))
```

Figure 15: **Element-match** takes a pattern variable, a datum to match, a dictionary, and **fail** and **succeed** continuations. If there is already an entry for the variable in the dictionary, the match succeeds if the value in the entry is the same as the datum to be matched. Otherwise the original dictionary is extended by adding a binding of the pattern variable name to the datum. The extended dictionary is passed to the **succeed** continuation.

```
(define (segment-match pat dat dict fail succeed)
  (let ((vname (var-name (car pat))) (p (var-restriction (car pat))))
    (let ((v (lookup vname dict)))
      (if (entry-exists? v)
          (let ((val (element-in v)))
            (if (restrict-segment p val)
                (let ((rest (after-initial-segment val dat)))
                  (if (not (eq? rest 'no-initial-segment))
                      (match (cdr pat) rest dict fail succeed)
                      (fail)))
                (fail)))
          (let ()                     ;to permit internal definition
            (define (try-segment rest)
              (define (try-longer-segment)
                (if (null? rest) (fail) (try-segment (cdr rest))))
              (if (restrict-segment p (make-segment dat rest))
                  (match (cdr pat)
                         rest
                         (extend-dictionary vname
                                            (make-segment dat rest)
                                            dict)
                         try-longer-segment
                         succeed)
                  (try-longer-segment)))
            (try-segment dat))))))
```

Figure 16: The procedure that handles segment matching implements backtracking via a failure continuation that tries successively longer segments. The utility procedure **after-initial-segment** returns the rest of the datum after a given initial segment, or notes that the initial segment of the datum does not match the desired segment.

```
(define (memoize f)
  (let ((table (empty-table)))
    (lambda (x)
      (let ((seen (lookup-in-table x table)))
        (if (valid-table-entry? seen)
            (table-entry-value seen)
            (let ((ans (f x)))
              (insert-in-table! table (make-table-entry x ans))
              ans))))))

(define fib
  (memoize
   (lambda (n)
     (cond ((= n 0) 0)
           ((= n 1) 1)
           (else (+ (fib (- n 1)) (fib (- n 2))))))))
```

Figure 17: **Memoize** is a higher-order procedure that transforms its argument procedure **f** into a new procedure that computes the same result, only avoiding redundant computation. The recursive **fib** procedure uses the memoizer to compute **(fib n)** in time proportional to n.

```
(define (make-simplifier the-rules)
  (define simplify-exp
    (memoize
     (lambda (exp)
       (let ((result
              (try-rules (if (compound? exp)
                             (map simplify-exp exp)
                             exp)
                         the-rules)))
         (if (equal? result exp)
             result
             (simplify-exp result))))))
  simplify-exp)
```

Figure 18: Memoizing the simplifier can greatly improve the performance of the rule interpreter.