repeatedly performing in-place transformations of the graph. **Combinator reduction** is a synonym. By contrast, the evaluation mechanism used in our interpreter is called the "SECD machine" approach, after Landin [1964].

lazy evaluation. An evaluation regime in which the arguments to functions are not evaluated until the corresponding formal parameters are actually used. **Delayed evaluation**, **normal-order evaluation**, and **call-by-need** are synonyms.

lazy list. An infinite list. Lazy lists are made possible by having `cons` be nonstrict.

strict function. A function which must evaluate all its arguments. An example of a function which is naturally strict is `+`. A function which is nonstrict is `if`. The difference between lazy evaluation and eager evaluation is that in eager evaluation certain functions — namely, `cons` and all user-defined functions — are *artificially* turned into strict functions.

thunk. A closure-like object which contains all the information necessary to evaluate an actual parameter or list component (namely, the expression and the environment) if and when such evaluation is indicated. The differences between a thunk and a closure are (1) a closure always represents a function, whereas a thunk can represent any type of value; and (2) a closure is evaluated when it is applied, whereas a thunk is evaluated when the formal parameter with which it is associated is referenced, or when the list position it occupies is accessed. **Suspension** is a synonym.

5.10.2 FURTHER READING

The literature on lazy evaluation is scattered, with no book-length treatment for the noninitiate. The discussion of streams in Abelson and Sussman [1985, Chapter 3] is excellent, and the articles by Hughes [1984] and Turner [1985a] are also recommended. Turner [1983] is the most recent manual for SASL. Kahn and MacQueen [1977] is a classic paper in which streams are used as a model for parallel processing.

It should be mentioned that SASL itself is now obsolete, having been superseded by the language MIRANDA, which can be described as SASL with type inference and pattern-matching (as in ML). Turner [1985b,1985c] are introductory papers on MIRANDA.

For further reading on graph reduction, we refer the reader to Turner [1979a], and especially to the recent book by Peyton Jones [1987], which will lead the interested reader to a wealth of research in this area.

λ-calculus is a language of enormous importance in the theoretical study of programming languages, which has for years been studied by mathematical logicians. Church [1941] is an accessible treatment from this school; Barendregt [1981] is the standard reference but is highly mathematical. λ-calculus has been revitalized in recent years by its applications in computer science, particularly in the formal definition of programming languages (what is known as *denotational semantics*). Gordon [1979] is a pragmatic treatment; Stoy [1977] and Schmidt [1986] cover the theory as well as the applications. Discussions that are less formal and emphasize λ-calculus as a programming language can be found in Burge [1975], Peyton Jones [1987], and Wegner [1968].

Vector addition systems were introduced in Karp and Miller [1969]. Our definition is simpler than the one given there in our requirement that the points in $\mathcal{P}$ have nonnegative coordinates. This makes it simple for us to decide when (x_0, y_0) is *not* reachable. If $\mathcal{P}$ is permitted to include points with negative coordinates, our method of finding paths still works when a path exists, but we can't know when to stop looking for a path. Furthermore, if we restore the restriction that the path remain entirely within the first quadrant, nobody knows if this problem is *decidable*, that is, if any computer program can be written which infallibly determines when (x_0, y_0) is *not* reachable.

5.11 Exercises

5.11.1 LEARNING ABOUT THE LANGUAGE

1. Construct the following lists by the method of recursion schemes (page 165).
   (a) The list `evens` containing all even numbers.
   (b) The list `xlist` from Section 5.2.3.
   (c) The list of all lists of binomial coefficients (see page 72).
   (d) The list of all Fibonacci numbers, that is, the list $0, 1, 1, 2, 3, 5, 8, \ldots$. Give the general solution for recursion schemes of the form:

$$\begin{aligned} x_0 &= \text{some initial value} \\ x_1 &= \text{some initial value} \\ x_{i+2} &= f(x_{i+1}, x_i). \end{aligned}$$

2. These questions relate to the implementation of ZF-expressions. You are to code the functions described in each part. Note first that the function `filter` (defined on page 165) implements the simplest kind of ZF-expression; namely, `(filter pred lis)` is equivalent to the ZF-expression:

```
{ a; a <- lis; pred a }.
```

   (a) `(filter-fun f pred lis)` corresponds to the ZF-expression

```
{ f a; a <- lis; pred a }
```

   (b) `(mkfinitepairs l1 l2)` forms all pairs of elements from `l1` and `l2`,

assuming l1 and l2 are finite. It implements the ZF-expression

```
{ a, b; a <- l1; b <- l2 }
```

for finite l1, l2.

(c) Actually, ZF-expressions are more general than we have indicated, in that l2 can be a function of a:

```
{ a, b; a <- l1; b <- (l2 a) }
```

denotes the list of all pairs (a, b) such that b is in the list (l2 a). For example,

```
{ m, n; m <- 1..5; n <- m..m+1 }
```

returns the list of pairs (1,1), (1,2), (2,2), (2,3), and so on (not necessarily in that order).

(mkfinitepairs* l1 l2) should implement this ZF-expression, assuming that l1 is a finite list and that (l2 a) is a finite list for all a.

(d) (mkpairs l1 l2) extends the ZF-expression from part (b) to *arbitrary* (finite or infinite) l1 and l2. (Since this case is much trickier than for finite lists, it is most unlikely that your solution to part (b) works for infinite lists.) Here is an approach to this problem. Suppose l1 = (x_1 x_2 ...), and l2 = (y_1 y_2 ...). Then: (1) generate the list

$$(((x_1\ y_1)\ (x_1\ y_2)\ \ldots)\ ((x_2\ y_1)\ (x_2\ y_2)\ \ldots)\ \ldots).$$

(2) Write a function merge, which merges two lists, finite or infinite, into a single list. (3) Use merge to write a function interleave which flattens a list of the form

$$((a_1\ a_2\ \ldots)\ (b_1\ b_2\ \ldots)\ \ldots),$$

where the sublists may be infinite. (Note that flatten does not work in this case.)

(e) Program mkpairs*, which implements ZF-expressions of the form

```
{ a, b; a <- l1; b <- (l2 a) }
```

in the case that l1 can be infinite and each (l2 a) can be infinite.

(f) (filter-pair pred l1 l2) implements

```
{ a, b; a <- l1; b <- l2; pred a b }
```


Use it to construct the list relative-primes containing every pair of integers greater than 2 which are relatively prime (have no common factors). Then program filter-pair*, which bears the same relation to filter-pair that mkpairs* does to mkpairs. Specifically, it implements the ZF-expression

```
{ a, b; a <- l1; b <- (l2 a); pred a b }.
```

(g) (filter-pair-fun f pred l1 l2) implements

```
{ f a b; a <- l1; b <- l2; pred a b }
```

Program filter-pair-fun and filter-pair-fun*. The latter can be used to code a nice version of permutations, given in Turner [1983]:

```
perms () = (())
perms x = { a:p; a <- x; p <- perms (x -- a) },
```

where "(x -- a)" is the list obtained by removing a from the list x.

3. These questions relate to the reachability problem (Section 5.5).

(a) Our reach-dfs and reach-bfs return only a truth value indicating the existence or nonexistence of a path. Modify them to return the path itself.

(b) The reachability problem in n dimensions deals with points in n-space; i.e., instead of points it uses lists of nonnegative integers of length n. Code reach-dfs and reach-bfs so that they can be applied to a reachability problem of any dimension.

(c) (reach-dfs p0 PATHS k) searches for a path to p0 of length not greater than k; it returns either T if it finds it, () if it can determine, by searching the tree to k levels, that no path exists, or Dunno if it determines that no path of length k or less exists but does not know whether a longer path might exist. It searches the k levels in depth-first order.

4. These problems relate to the λ-calculus (Section 5.8).

(a) Define FIB in λ-calculus, using the same technique as for FAC.

(b) Modify the definition of integers to allow for negative numbers, using a "signed-magnitude" representation.

(c) Show how Y can be extended to an arbitrarily long list of mutually-recursive functions, i.e., a letrec.