10 Functional Languages

10.6 Theoretical Foundations

Mathematically, a function is a single-valued mapping: it associates every element in one set (the domain) with (at most) one element in another set (the range). In conventional notation, we indicate the domain and range by writing

$$\text{sqrt : } \mathbb{R} \rightarrow \mathbb{R}$$

We can of course, have functions of more than one variable—that is, functions whose domains are Cartesian products:

$$\text{plus : } [\mathbb{R} \times \mathbb{R}] \rightarrow \mathbb{R}$$

If a function provides a mapping for every element of the domain, the function is said to be total. Otherwise, it is said to be partial. Our sqrt function is partial: it does not provide a mapping for negative numbers. We could change our definition to make the domain of the function the non-negative numbers, but such changes are often inconvenient, or even impossible: inconvenient because we should like all mathematical functions to operate on $\mathbb{R}$; impossible because we may not know which elements of the domain have mappings and which do not. Consider for example the function $f$ that maps every natural number $a$ to the smallest natural number $b$ such that the digits of the decimal representation of $a$ appear $b$ digits to the right of the decimal point in the decimal expansion of $\pi$. Clearly $f(59) = 4$, because $\pi = 3.14159 \ldots$ But what about $f(428945028)$, or in general $f(n)$ for arbitrary $n$? Absent results from number theory, it is not at all clear how to characterize the values at which $f$ is defined. In such a case a partial function is essential.

It is often useful to characterize functions as sets or, more precisely, as subsets of the Cartesian product of the domain and the range:

$$\text{sqrt } \subset [\mathbb{R} \times \mathbb{R}]$$

$$\text{plus } \subset [\mathbb{R} \times \mathbb{R} \times \mathbb{R}]$$
We can specify which subset using traditional set notation:

\[
\text{sqrt} \equiv \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid y > 0 \land x = y^2\}
\]

\[
\text{plus} \equiv \{(x, y, z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \mid z = x + y\}
\]

Note that this sort of definition tells us what the value of a function like sqrt is, but it does not tell us how to compute it; more on this distinction below.

One of the nice things about the set-based characterization is that it makes it clear that a function is an ordinary mathematical object. We know that a function from A to B is a subset of \(A \times B\). This means that it is an element of the powerset of \(A \times B\)—the set of all subsets of \(A \times B\), denoted \(2^{A \times B}\):

\[
\text{sqrt} \in 2^{\mathbb{R} \times \mathbb{R}}
\]

Similarly

\[
\text{plus} \in 2^{\mathbb{R} \times \mathbb{R} \times \mathbb{R}}
\]

Note the overloading of notation here. The powerset \(2^A\) should not be confused with exponentiation, though it is true that for a finite set \(A\) the number of elements in the powerset of \(A\) is \(2^n\), where \(n = |A|\), the cardinality of \(A\).

Because functions are single-valued, we know that they constitute only some of the elements of \(2^{A \times B}\). Specifically, they constitute all and only those sets of pairs in which the first component of each pair is unique. We call the set of such sets the function space of \(A\) into \(B\), denoted \(A \rightarrow B\). Note that \((A \rightarrow B) \subset 2^{A \times B}\). In our examples:

\[
\text{sqrt} \in [\mathbb{R} \rightarrow \mathbb{R}]
\]

\[
\text{plus} \in [((\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R})]
\]

Now that functions are elements of sets, we can easily build higher-order functions:

\[
\text{compose} \equiv \{(f, g, h) \mid \forall x \in \mathbb{R}, h(x) = f(g(x))\}
\]

What are the domain and range of compose? We know that \(f, g,\) and \(h\) are elements of \(\mathbb{R} \rightarrow \mathbb{R}\). Thus

\[
\text{compose} \in [(\mathbb{R} \rightarrow \mathbb{R}) \times (\mathbb{R} \rightarrow \mathbb{R})] \rightarrow (\mathbb{R} \rightarrow \mathbb{R})
\]

Note the similarity to the notation employed by the ML type inference system (Section 7.2.4).

Using the notion of “currying” from Section 10.5, we note that there is an alternative characterization for functions like plus. Rather than a function from pairs of reals to reals, we can capture it as a function from reals to functions from reals to reals:

\[
\text{curried}_\text{plus} \in \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})
\]
We shall have more to say about currying in Section 10.6.3.

### 10.6.1 Lambda Calculus

As we suggested in the main text, one of the limitations of the function-as-set notation is that it is nonconstructive: it doesn’t tell us how to compute the value of a function at a given point (i.e., on a given input). Church designed the lambda calculus to address this limitation. In its pure form, lambda calculus represents everything as a function. The natural numbers, for example, can be represented by a distinguished zero function (commonly the identity function) and a successor function. (One common formulation uses a select, second function that takes two arguments and returns the second of them. The successor function is then defined in such a way that the number \( n \) ends up being represented by a function that, when applied to select, second \( n \) times, returns the identity function [Mic89, Sec. 3.5; Sta95, Sec. 7.6]; see Exercise 10.21.) While of theoretical importance, this formulation of arithmetic is highly cumbersome. We will therefore take ordinary arithmetic as a given in the remainder of this subsection. (And of course all practical functional programming languages provide built-in support for both integer and floating-point arithmetic.)

A lambda expression can be defined recursively as (1) a name; (2) a lambda abstraction consisting of the letter \( \lambda \), a name, a dot, and a lambda expression; (3) a function application consisting of two adjacent lambda expressions; or (4) a parenthesized lambda expression. To accommodate arithmetic, we will extend this definition to allow numeric literals.

When two expressions appear adjacent to one another, the first is interpreted as a function to be applied to the second:

\[
\text{sqrt } n
\]

Most authors assume that application associates left-to-right (so \( f A B \) is interpreted as \( (f A) B \), rather than \( f (A B) \)), and that application has higher precedence than abstraction (so \( \lambda x. A B \) is interpreted as \( \lambda x. (A B) \), rather than \( (\lambda x. A) B \)). ML adopts these rules.

Parentheses are used as necessary to override default groupings. Specifically, if we distinguish between lambda expressions that are used as functions and those that are used as arguments, then the following unambiguous CFG can be used to generate lambda expressions with a minimal number of parentheses:

\[
\begin{align*}
\text{expr} & \rightarrow \text{name} \mid \text{number} \mid \lambda \text{name} . \text{expr} \mid \text{func} \text{arg} \\
\text{func} & \rightarrow \text{name} \mid \text{( } \lambda \text{name} \ . \text{expr} \text{ ) } \mid \text{func arg} \\
\text{arg} & \rightarrow \text{name} \mid \text{number} \mid \text{( } \lambda \text{name} \ . \text{expr} \text{ ) } \mid \text{func arg}
\end{align*}
\]

In words: we use parentheses to surround an abstraction that is used as either a function or an argument, and around an application that is used as an argument.
The letter \( \lambda \) introduces the lambda calculus equivalent of a formal parameter. The following lambda expression denotes a function that returns the square of its argument:

\[
\lambda x. \text{times } x \ x
\]

The name (variable) introduced by a \( \lambda \) is said to be *bound* within the expression following the dot. In programming language terms, this expression is the variable's scope. A variable that is not bound is said to be *free*.

As in a lexically scoped programming language, a free variable needs to be defined in some surrounding scope. Consider, for example, the expression

\[
\lambda x. \lambda y. \text{times } x \ y.
\]

In the inner expression \( (\lambda y. \text{times } x \ y) \), \( y \) is bound but \( x \) is free. There are no restrictions on the use of a bound variable: it can play the role of a function, an argument, or both. Higher-order functions are therefore completely natural.

If we wish to refer to them later, we can give expressions names:

\[
\begin{align*}
\text{square} & \equiv \lambda x. \text{times } x \ x \\
\text{identity} & \equiv \lambda x. x \\
\text{const7} & \equiv \lambda x.7 \\
\text{hypot} & \equiv \lambda x.\lambda y. \text{sqrt} \ (\text{plus} \ (\text{square } x) \ (\text{square } y))
\end{align*}
\]

Here \( \equiv \) is a metasymbol meaning, roughly, “is an abbreviation for.”

To compute with the lambda calculus, we need rules to evaluate expressions. It turns out that three rules suffice:

**beta reduction**: For any lambda abstraction \( \lambda x. E \) and any expression \( M \), we say

\[
(\lambda x. E) \ M \rightarrow_\beta E[M/x]
\]

where \( E[M/x] \) denotes the expression \( E \) with all free occurrences of \( x \) replaced by \( M \). Beta reduction is not permitted if any free variables in \( M \) would become bound in \( E[M/x] \).

**alpha conversion**: For any lambda abstraction \( \lambda x. E \) and any variable \( y \) that has no free occurrences in \( E \), we say

\[
\lambda x. E \rightarrow_\alpha \lambda y. E[y/x]
\]

**eta reduction**: A rule to eliminate “surplus” lambda abstractions. For any lambda abstraction \( \lambda x. E \), where \( E \) is of the form \( F \ x \), and \( x \) has no free occurrences in \( F \), we say

\[
\lambda x. F \ x \rightarrow_\eta F
\]
10.6.1 Lambda Calculus

Figure 10.3 Reduction of a lambda expression. The top line consists of a function applied to three arguments. The first argument (underlined) is the "select first" function, which takes two arguments and returns the first. The second argument is the symbol $h$, which must be either a constant or a variable bound in some enclosing scope (not shown). The third argument is an "apply to self" function that takes one argument and applies it to itself. The particular series of reductions shown occurs in normal order: It terminates with a simplest (normal) form of simply $h$.

Example 10.56

Delta reduction for arithmetic

To accommodate arithmetic we will also allow an expression of the form $\text{op} \ x \ y$, where $x$ and $y$ are numeric literals and $\text{op}$ is one of a small set of standard functions, to be replaced by its arithmetic value. This replacement is called delta reduction. In our examples we will need only the functions plus, minus, and times:

\[
\begin{align*}
\text{plus} \ 2 \ 3 & \rightarrow_\delta 5 \\
\text{minus} \ 5 \ 2 & \rightarrow_\delta 3 \\
\text{times} \ 2 \ 3 & \rightarrow_\delta 6
\end{align*}
\]

Beta reduction resembles the use of call by name parameters (Section 8.3.1). Unlike Algol 60, however, the lambda calculus provides no way for an argument to carry its referencing environment with it; hence the requirement that an argument not move a variable into a scope in which its name has a different meaning. Alpha conversion serves to change names to make beta reduction possible. Eta reduction is comparatively less important. If \textit{square} is defined as above, eta reduction allows us to say that

\[
\lambda x. \text{square} \ x \rightarrow_\eta \text{square}
\]

In English, \textit{square} is a function that squares its argument; $\lambda x. \text{square} \ x$ is a function of $x$ that squares $x$. The latter reminds us explicitly that it’s a function (i.e., that it takes an argument), but the former is a little less messy looking.

Example 10.57

Eta reduction

Through repeated application of beta reduction and alpha conversion (and possibly eta reduction), we can attempt to reduce a lambda expression to its simplest possible form—a form in which no further beta reductions are possible. An example can be found in Figure 10.3. In line (2) of this derivation we have to

Example 10.58

Reduction to simplest form

Copyright © 2009 by Elsevier Inc. All rights reserved.
employ an alpha conversion because the argument that we need to substitute for 
g contains a free variable \(h\) that is bound within \(g\)’s scope. If we were to make 
the substitution of line (3) without first having renamed the bound \(h\) (as \(k\)), then 
the free \(h\) would have been \textit{captured}, erroneously changing the meaning of the 
expression.

In line (5) of the derivation, we had a choice as to which subexpression to 
reduce. At that point the expression as a whole consisted of a function application 
in which the argument was itself a function application. We chose to substi-
tute the main argument \(\left(\lambda x.x x\right) \left(\lambda x.x x\right)\), unevaluated, into the body of the 
main lambda abstraction. This choice is known as \textit{normal-order} reduction, and 
corresponds to normal-order evaluation of arguments in programming languages, 
as discussed in Sections 6.6.2 and 10.4. In general, whenever more than one beta 
reduction could be made, normal order chooses the one whose \(\lambda\) is left-most in 
the overall expression. This strategy substitutes arguments into functions before 
reducing them. The principal alternative, \textit{applicative-order} reduction, reduces 
both the function part and the argument part of every function application to 
the simplest possible form before substituting the latter into the former. 

Church and Rosser showed in 1936 that simplest forms are unique: any series 
of reductions that terminates in a nonreducible expression will produce the same 
result. Not all reductions terminate, however. In particular, there are expres-
sions for which no series of reductions will terminate, and there are others in 
which normal-order reduction will terminate but applicative-order reduction will 
not. The example expression of Figure 10.3 leads to an infinite “computation” 
under applicative-order reduction. To see this, consider the expression at line 
(5). This line consists of the constant function \(\left(\lambda y.h\right)\) applied to the argument 
\(\left(\lambda x.x x\right) \left(\lambda x.x x\right)\). If we attempt to evaluate the argument before substituting it 
into the function, we run through the following steps:

\[
\begin{align*}
\left(\lambda x.x x\right) \left(\lambda x.x x\right) \\
\rightarrow_\beta \left(\lambda x.x x\right) \left(\lambda x.x x\right) \\
\rightarrow_\beta \left(\lambda x.x x\right) \left(\lambda x.x x\right) \\
\rightarrow_\beta \left(\lambda x.x x\right) \left(\lambda x.x x\right) \\
\ldots
\end{align*}
\]

In addition to showing the uniqueness of simplest (normal) forms, Church and 
Rosser showed that if any evaluation order will terminate, normal order will. This 
pair of results is known as the \textit{Church-Rosser theorem}.

\subsection{Control Flow}

We noted at the beginning of the previous subsection that arithmetic can be 
modeled in the lambda calculus using a distinguished zero function (commonly 
the identity) and a successor function. What about control-flow constructs— 
selection and recursion in particular?
### Example 10.60

Booleans and conditionals

The select\_first function, \( \lambda x.\lambda y.x \), is commonly used to represent the Boolean value true. The select\_second function, \( \lambda x.\lambda y.y \), is commonly used to represent the Boolean value false. Let us denote these by \( T \) and \( F \). The nice thing about these definitions is that they allow us to define an if function very easily:

\[
\text{if } \equiv \lambda c.\lambda t.\lambda e.\text{cte}
\]

Consider:

\[
\begin{align*}
\text{if } T \ 34 & \equiv (\lambda c.\lambda t.\lambda e.\text{cte})(\lambda x.\lambda y.x)\ 34 \\
& \rightarrow^*_{\beta} (\lambda x.\lambda y.x)\ 34 \\
& \rightarrow^*_{\beta} 3 \\
\text{if } F \ 34 & \equiv (\lambda c.\lambda t.\lambda e.\text{cte})(\lambda x.\lambda y.y)\ 34 \\
& \rightarrow^*_{\beta} (\lambda x.\lambda y.y)\ 34 \\
& \rightarrow^*_{\beta} 4 \\
\end{align*}
\]

Functions like equal and greater\_than can be defined to take numeric values as arguments, returning \( T \) or \( F \).

Recursion is a little tricky. An equation like

\[
gcd \equiv \lambda a.\lambda b.(\text{if } (\text{equal } a\ b)\ a \\
(\text{if } (\text{greater\_than } a\ b)\ (\text{gcd } (\text{minus } a\ b)\ b\ (\text{gcd } (\text{minus } b\ a)\ a)))
\]

is not really a definition at all, because gcd appears on both sides. Our previous definitions (\( T, F, \text{if} \)) were simply shorthand: we could substitute them out to obtain a pure lambda expression. If we try that with gcd, the “definition” just gets bigger, with new occurrences of the gcd name. To obtain a real definition, we first rewrite our equation using beta abstraction (the opposite of beta reduction):

\[
gcd \equiv (\lambda g.\lambda a.\lambda b.(\text{if } (\text{equal } a\ b)\ a \\
(\text{if } (\text{greater\_than } a\ b)\ (g (\text{minus } a\ b)\ b\ (g (\text{minus } b\ a)\ a))))))
gcd
\]

Now our equation has the form

\[
gcd \equiv f\ gcd
\]

where \( f \) is the perfectly well-defined (nonrecursive) lambda expression

\[
\lambda g.\lambda a.\lambda b.(\text{if } (\text{equal } a\ b)\ a \\
(\text{if } (\text{greater\_than } a\ b)\ (g (\text{minus } a\ b)\ b\ (g (\text{minus } b\ a)\ a))))
\]

Clearly gcd is a fixed point of \( f \).
As it turns out, for any function \( f \) given by a lambda expression, we can find the least fixed point of \( f \), if there is one, by applying the fixed-point combinator
\[
\lambda h. (\lambda x.h(xx)) (\lambda x.h(xx))
\]
commonly denoted \( Y \). \( Y \) has the property that for any lambda expression \( f \), if the normal-order evaluation of \( Yf \) terminates, then \( f(Yf) \) and \( Yf \) will reduce to the same simplest form (see Exercise \( \S \)10.9). In the case of our \( \text{gcd} \) function, we have
\[
\text{gcd} \equiv (\lambda h. (\lambda x.h(xx)) (\lambda x.h(xx))) \\
(\lambda g.\lambda a.\lambda b. (\text{if } (\text{equal } ab) a) \\
(\text{if } (\text{greater than } ab) g(\text{minus } ab) b) (g(\text{minus } ba) a)))
\]
Figure \( \S \)10.4 traces the evaluation of \( \text{gcd} \ 4 \ 2 \). Given the existence of the \( Y \) combinator, most authors permit recursive “definitions” of functions, for convenience.

### 10.6.3 Structures

Just as we can use functions to build numbers and truth values, we can also use them to encapsulate values in structures. Using Scheme terminology for the sake of clarity, we can define simple list-processing functions as follows:

- \( \text{cons} \equiv \lambda a.\lambda d.\lambda x. x a d \)
- \( \text{car} \equiv \lambda l.\text{select}_\text{first} \)
- \( \text{cdr} \equiv \lambda l.\text{select}_\text{second} \)
- \( \text{nil} \equiv \lambda x. T \)
- \( \text{null?} \equiv \lambda l. (\lambda x. (\lambda y. F)) l \)

where \( \text{select}_\text{first} \) and \( \text{select}_\text{second} \) are the functions \( \lambda x.\lambda y. x \) and \( \lambda x.\lambda y. y \), respectively—functions we also use to represent \text{true} and \text{false}.

Using these definitions we can see that
\[
\text{car}(\text{cons } A B) \equiv (\lambda l.l \text{select}_\text{first}) (\text{cons } A B) \\
\rightarrow^\beta (\text{cons } A B) \text{ select}_\text{first} \\
\equiv ((\lambda a.\lambda d.\lambda x.a d) A B) \text{ select}_\text{first} \\
\rightarrow^*^\beta (\lambda x.A B) \text{ select}_\text{first} \\
\rightarrow^\beta \text{ select}_\text{first} A B \\
\equiv (\lambda x.\lambda y.x) A B \\
\rightarrow^*^\beta A
\]
10.6.3 Structures

\[ \text{gcd} \ 24 \equiv Yf \ 24 \]
\[ \equiv ((\lambda h.(\lambda x.h(x)))(\lambda x.h(x)))f \ 24 \]
\[ \rightarrow_\beta ((\lambda x.f(x))(\lambda x.f(x))) \ 24 \]
\[ \equiv (k \ k) \ 24, \ \text{where} \ k \equiv \lambda x.f(x) \]
\[ \rightarrow_\beta (f(k \ k)) \ 24 \]
\[ \equiv ((\lambda g.\lambda a.\lambda b.(if \ (= \ a \ b) \ a \ (if \ (> \ a \ b) \ (g(-a \ b) \ b) \ (g(b \ a) \ a))) \ (k \ k)) \ 24 \]
\[ \rightarrow_\beta (\lambda a.\lambda b.(if \ (= \ a \ b) \ a \ (if \ (> \ a \ b) \ (k \ k)(-a \ b) \ b) \ (k(k)(-b \ a) \ a))) \ 24 \]
\[ \rightarrow_\beta^* \text{if} \ (= \ 24) \ 2 \ (if \ (> \ 24) \ ((k \ k)(-24)4) \ ((k \ k)(-42)2)) \]
\[ \equiv (\lambda c.\lambda t.\lambda e.\lambda e.\epsilon) \ 2 \ (if \ (> \ 24) \ ((k \ k)(-24)4) \ ((k \ k)(-42)2)) \]
\[ \rightarrow_\beta^* \text{if} \ (> \ 24) \ ((k \ k)(-24)4) \ ((k \ k)(-42)2) \]
\[ \rightarrow \ldots \]
\[ \rightarrow (k \ k)(-42)2 \]
\[ \equiv ((\lambda x.f(x))k)(-42)2 \]
\[ \rightarrow_\beta (f(k \ k))(-42)2 \]
\[ \equiv ((\lambda g.\lambda a.\lambda b.(if \ (= \ a \ b) \ a \ (if \ (> \ a \ b) \ (g(-a \ b) \ b) \ (g(b \ a) \ a))) \ (k \ k))(-42)2 \]
\[ \rightarrow_\beta (\lambda a.\lambda b.(if \ (= \ a \ b) \ a \ (if \ (> \ a \ b) \ (k \ k)(-a \ b) \ b) \ (k(k)(-b \ a) \ a)))(-42)2 \]
\[ \rightarrow_\beta^* \text{if} \ (= \ -42)2 \ (-42) \ (if \ (> \ -42)2) \ ((k \ k)(-(-42)2)2) \ ((k \ k)(-2(-42))(-42)) \]
\[ \equiv (\lambda c.\lambda t.\lambda e.\epsilon) \]
\[ \quad (= \ (-42)2) \ (-42) \ (if \ (> \ -42)2) \ ((k \ k)(-(-42)2)2) \ ((k \ k)(-2(-42))(-42)) \]
\[ \rightarrow_\beta^* \text{if} \ (= \ -42)2 \ (-42) \ (if \ (> \ -42)2) \ ((k \ k)(-(-42)2)2) \ ((k \ k)(-2(-42))(-42)) \]
\[ \rightarrow_\delta \text{if} \ (> \ -42)2 \ ((k \ k)(-(-42)2)2) \ ((k \ k)(-2(-42))(-42)) \]
\[ \equiv (\lambda x.\lambda y.x)(-42) \ (if \ (> \ -42)2) \ ((k \ k)(-(-42)2)2) \ ((k \ k)(-2(-42))(-42)) \]
\[ \rightarrow_\beta^* (-42) \]
\[ \rightarrow_\delta 2 \]

Figure 10.4 Evaluation of a recursive lambda expression. As explained in the body of the text, \text{gcd} is defined to be the fixed-point combinator \( Y \) applied to a beta abstraction \( f \) of the standard recursive definition for greatest common divisor. Specifically, \( Y \) is \( \lambda h.((\lambda x.h(x)))(\lambda x.h(x)) \) and \( f \) is \( \lambda g.\lambda a.\lambda b.(if \ (= \ a \ b) \ a \ (if \ (> \ a \ b) \ (g(-a \ b) \ b) \ (g(b \ a) \ a))) \). For brevity we have used \( =, >, \) and \( - \) in place of equal, greater_than, and minus. We have performed the evaluation in normal order.
Because every lambda abstraction has a single argument, lambda expressions are naturally curried. We generally obtain the effect of a multiargument function by nesting lambda abstractions:

$$
\text{compose} \equiv \lambda f. \lambda g. \lambda x. f \, g \, x
$$

which groups as

$$
\lambda f. (\lambda g. (\lambda x. (f \, g) \, x))
$$

We commonly think of compose as a function that takes two functions as arguments and returns a third function as its result. We could just as easily, however, think of compose as a function of three arguments: the $f$, $g$, and $x$ above. The official story, or course, is that compose is a function of one argument that evaluates to a function of one argument that in turn evaluates to a function of one argument.

If desired, we can use our structure-building functions to define a noncurried version of compose whose (single) argument is a pair:

$$
\text{paired-compose} \equiv \lambda p. \lambda x. (\text{car} \, p) \, ((\text{cdr} \, p) \, x)
$$
If we consider the pairing of arguments as a general technique, we can write a `curry` function that reproduces the single-argument version, just as we did in Scheme in Section 10.5:

\[
\text{curry} \equiv \lambda f. (\lambda a. (\lambda b. f(\text{cons} \ a \ b)))
\]

**CHECK YOUR UNDERSTANDING**

22. What is the difference between *partial* and *total* functions? Why is the difference important?

23. What is meant by the *function space* \( A \rightarrow B \)?

24. Define *beta reduction*, *alpha conversion*, *eta reduction*, and *delta reduction*.

25. How does beta reduction in lambda calculus differ from lazy evaluation of arguments in a nonstrict programming language like Haskell?

26. Explain how lambda expressions can be used to represent Boolean values and control flow.

27. What is *beta abstraction*?

28. What is the *Y* combinator? What useful property does it possess?

29. Explain how lambda expressions can be used to represent structured values such as lists.

30. State the *Church-Rosser theorem*. 

Copyright © 2009 by Elsevier Inc. All rights reserved.